Aegir

Date: July 1, 1997

PROGRESS REPORT

on the Project
Automatic Target Recognition (N94-124)

for
SMALL BUSINESS INNOVATION RESEARCH
Phase II

Aegir Contract No.: 1079-000
Contract No.: N00014-96-C-0069
CDRL No. 0001

Prepared by:

Filimage, Inc. for Aegir Systems, Inc.
2051 N. Solar Drive; Suite 200
Oxnard, CA 93030
(805)485-4888

for
Program Officer
Office of Naval Research
800 North Quincy Street
Arlington, Virginia 22217-5660
Attn: Julia Abrahams Code 311
Ref: Contract N0014-96-C-0069

Prepared by:

B. Rozovsky
Author

Approved by:

Program Manager

Data Manager

1 Introduction

Filtering, prediction, and smoothing (FPS) are the three basic components of the data assimilation process in target tracking. An analytical solution of the FPS problem is possible only in a handful of particular cases, the most important of which is linear. In this case the solution is given by the Kalman filter. However, in many important cases, such as passive sonar, radar warning systems, infrared search and track, the systems are generically nonlinear. To date, the extended Kalman filter (EKF) has been the dominant algorithm technology in real-time estimation, tracking, and similar applications. A major reason for its success has been the fact that it has offered a reasonable compromise between real-time operation and satisfactory performance in some nonlinear problems. On the other hand, the EKF is a completely heuristic algorithm, requires readjustment to each particular problem, and is unstable in nonlinear problems which involve jumps, maneuvers, etc.

Nonlinear filtering is the process of computing estimates of the current state $\boldsymbol{x}_t$ of a nonlinear dynamic system (e.g., a rapidly maneuvering target), given current measurements (which are nonlinear functions of the system state) together with some degree of knowledge of the states of the system at previous instants. The state may include such unknown target characteristics as position, speed, acceleration, aspect angle, etc. Kinematic data $\boldsymbol{z}_k$ are collected at discrete time instants $k = 0, 1, 2, \ldots$. The relationship between measurements and target states is modeled by a (generally nonlinear) measurement equation of the form $\boldsymbol{z}_k = h(\boldsymbol{x}_k, \boldsymbol{v}_k)$ where $\boldsymbol{v}_k$ denotes the noise process. The expected range of possible behaviors of the target is modeled by Markov-state transition equations of the form $\boldsymbol{x}_t = b(\boldsymbol{x}_t, \boldsymbol{w}_t)$ where $\boldsymbol{w}_t$ is another noise process.

In many practical situations the EKF and its variants do not perform well when the model functions b and h are highly nonlinear. The reason is that the EKF-like methods approximate the true posterior probability densities $p_{k|k}(\boldsymbol{x}_k \mid \boldsymbol{Z}^k)$ of the target state by Gaussian laws ($\boldsymbol{Z}^k = (\boldsymbol{z}_1, \ldots, \boldsymbol{z}_k)$ being the data observed up to time k). When a target executes a maneuver, $p_{k|k}(\boldsymbol{x}_k \mid \boldsymbol{Z}^k)$ often becomes multi-modal, with different modes corresponding to the most likely time-varying alternatives of the current target state. As data are collected one of these modes will eventually dominate the others, corresponding to the actual state of the target. By contrast, EKF is forced to approximate $p_{k|k}(\boldsymbol{x}_k \mid \boldsymbol{Z}^k)$ with a Gaussian density which is unimodal by definition. If the unique mode of the density fails to switch from the dominant pre-turn mode to the dominant post-turn mode, then the EKF will fail to hold track on the maneuvering target. Much better performance would result if the full nonlinear problem could be solved in real time.

In this report, we propose some new algorithms for the target tracking which are based on the spectral nonlinear FPS and the method of operator splitting. It has been demonstrated that these techniques decrease prediction error and provide a more accurate representation of the target bearings as compared to EKF. On the other hand, these algorithms are also fast, in that their calculations need only $O(N)$ flops per time step where N is the number of points in the spatial domain. That is, our approximation of the optimal nonlinear filter has an optimal computational complexity for arbitrary nonlinear systems.

In Section 2, the problem of target tracking is formulated in a proper framework of nonlinear filtering by using the basic models of manuvering targets. Section 3 is devoted to the development of the fast nonlinear filters, and a numerical example is given in Section 4.

2 Target Tracking via Nonlinear Filtering

Changes in target orientation is one of the most important sources of variability of its signature. The accuracy and the speed of target identification algorithms depend crucially on the availability and quality of target orientation data. In this Section we will introduce a new model for tracking dynamically changing orientation via optimal filtering. We will investigate the flight dynamics equations, develop the optimal target tracking and orientation estimation model, and conduct a preliminary study of the angle-only tracking problem.

2.1 Basic Models for Maneuvering Targets

Let $v = (v_1, v_2, v_3)$ and $q = (q_1, q_2, q_3)$ be the velocity and angular velocity of the target, respectively. Let $f = (f_1, f_2, f_3)$ be the relative time-derivative of the velocity, i.e., the rate of change of the velocity referred to the body-frame coordinates (with the rate of change being resolved back into the inertial coordinates).

In this case, we obtain the following form of Euler's equations (see [24])

$$\begin{aligned} \dot{v}_1 + q_3 v_2 - q_2 v_3 &= f_1, \\ \dot{v}_2 + q_1 v_3 - q_3 v_1 &= f_2, \\ \dot{v}_3 + q_2 v_1 - q_1 v_2 &= f_3. \end{aligned} \tag{1}$$

An important type of basic motion in target maneuvering is constant turn. In a constant turn, the velocity as viewed from the body-fixed frame is constant. So $f = 0$. Also q is independent of time t. Then equation (1) becomes

$$\begin{bmatrix} \dot{v}_1(t) \\ \dot{v}_2(t) \\ \dot{v}_3(t) \end{bmatrix} = \begin{bmatrix} 0 & -q_3 & q_2 \\ q_3 & 0 & -q_1 \\ -q_2 & q_1 & 0 \end{bmatrix} \begin{bmatrix} v_1(t) \\ v_2(t) \\ v_3(t) \end{bmatrix} . \tag{2}$$

In the notation of cross product of vectors, this may be simply written as $\dot{v}(t) = q \times v(t)$. Differentiating both sides with respect to t yields

$$\begin{aligned} \ddot{v}(t) &= q \times \dot{v}(t) \\ &= q \times (q \times v(t)) \\ &= q(q^T v(t)) - v(t)(q^T q) . \end{aligned}$$

Since the velocity and the angular velocity are perpendicular to each other, $q^T v(t) = 0$. Thus we end up with another *constant turn model*

$$\ddot{v}(t) = -\omega^2 v(t) , \tag{3}$$

where $\omega = ||q||_2 = \sqrt{q^T q}$ is the turn rate or turning speed. From $\dot{v} = q \times v$ and $q \perp v$, it follows that $||q||_2 = ||\dot{v}||_2 / ||v||_2$. This is a general formula for ω in the case $f = 0$.

Remark 1. Sometimes constant turn models are also called *constant speed turning models*. In fact, since the rotation matrix R is orthogonal and velocity $V(= Rv)$ in the body-frame coordinates is constant, we have $||v||_2 = ||R^T V||_2 = ||V||_2$, i.e., the speed is constant.

Remark 2. Comparing the two constant turn models, we see that (3) is formally more general than (2): In (2) both the direction and magnitude of the angular velocity are fixed, whereas in (3) only the turning speed is constant but the turning direction may still change. Also, model (3) is decoupled – each component has a separate equation independent of the other components, while model (2) is not. On the other hand, (2) is linear in q, while (3) is nonlinear in ω.

For practical applications, the above deterministic models are restrictive. By considering some additional white-noise acceleration besides the constant turn, we obtain from (2) the following three-dimensional *nearly constant turn model*:

$$\begin{bmatrix} \dot{v}_1(t) \\ \dot{v}_2(t) \\ \dot{v}_3(t) \end{bmatrix} = \begin{bmatrix} 0 & -q_3 & q_2 \\ q_3 & 0 & -q_1 \\ -q_2 & q_1 & 0 \end{bmatrix} \begin{bmatrix} v_1(t) \\ v_2(t) \\ v_3(t) \end{bmatrix} + \begin{bmatrix} \sigma_1 \dot{w}_1(t) \\ \sigma_2 \dot{w}_2(t) \\ \sigma_3 \dot{w}_3(t) \end{bmatrix}, \tag{4}$$

where σ_1, σ_2, and σ_3 are scaling parameters, and (w_1, w_2, w_3) is a standard three-dimensional Wiener process. This is a special case of the general equation (1), where the angular velocity q is constant and the "relative" acceleration f is assumed to be white noise.

Let $\Delta = t_{k+1} - t_k$ be the sampling period. Solving the deterministic part of (4) for v on the interval $[t_k, t_{k+1}]$, we obtain its discretized state equations:

$$\begin{aligned} \begin{bmatrix} v_1(t_{k+1}) \\ v_2(t_{k+1}) \\ v_3(t_{k+1}) \end{bmatrix} &= \begin{bmatrix} n_1^2 C_1 + C & n_1 n_2 C_1 - n_3 S & n_3 n_1 C_1 + n_2 S \\ n_1 n_2 C_1 + n_3 S & n_2^2 C_1 + C & n_2 n_3 C_1 - n_2 S \\ n_3 n_1 C_1 - n_2 S & n_2 n_3 C_1 + n_1 S & n_3^2 C_1 + C \end{bmatrix} \begin{bmatrix} v_1(t_k) \\ v_2(t_k) \\ v_3(t_k) \end{bmatrix} \\ &+ \begin{bmatrix} \sigma_1 \Delta w_{1,k+1} \\ \sigma_2 \Delta w_{2,k+1} \\ \sigma_3 \Delta w_{3,k+1} \end{bmatrix}, \quad \text{with} \begin{bmatrix} S \\ C \\ C_1 \end{bmatrix} = \begin{bmatrix} \sin \omega\Delta \\ \cos \omega\Delta \\ 1 - \cos \omega\Delta \end{bmatrix}. \end{aligned} \tag{5}$$

Here $n_i = q_i/\omega$, $\Delta w_{i,k+1} = w_i(t_{k+1}) - w_i(t_k)$ $(i = 1, 2, 3)$, and the stochastic noise term has been simplified. (An exact solution may be obtained but will be much more complex. For general results on exact discretization of linear stochastic systems, see [21].) The above formula is useful for describing the trajectory of a target when $q = \omega(n_1, n_2, n_3)^T$ is known. The deterministic part is also used in computer graphics for rotation models.

Now for the second constant turn model, if we add some white noise to system (3) and augment it by the identities $\dot{x}_i(t) = v_i(t)$ and $\dot{v}_i(t) = a_i(t)$, we get

$$\begin{bmatrix} \dot{x}_i(t) \\ \dot{v}_i(t) \\ \dot{a}_i(t) \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & -\omega^2 & 0 \end{bmatrix} \begin{bmatrix} x_i(t) \\ v_i(t) \\ a_i(t) \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \sigma_i \dot{w}_i(t), \quad i = 1, 2, 3, \tag{6}$$

where $x(t) = (x_1(t), x_2(t), x_3(t))$ is the position of the target and $a = (a_1, a_2, a_3)^T$ is its total acceleration in the inertial coordinates. Solving for $(x_i, v_i, a_i)^T$ on interval $[t_k, t_{k+1}]$, we obtain the following discrete-time *nearly constant speed turning model* (see [4]):

$$\begin{bmatrix} x_i(t_{k+1}) \\ v_i(t_{k+1}) \\ a_i(t_{k+1}) \end{bmatrix} = \begin{bmatrix} 1 & \frac{\sin \omega\Delta}{\omega} & \frac{1-\cos \omega\Delta}{\omega^2} \\ 0 & \cos \omega\Delta & \frac{\sin \omega\Delta}{\omega} \\ 0 & -\omega \sin \omega\Delta & \cos \omega\Delta \end{bmatrix} \begin{bmatrix} x_i(t_k) \\ v_i(t_k) \\ a_i(t_k) \end{bmatrix} + \begin{bmatrix} \frac{1}{6}\Delta^2 \\ \frac{1}{2}\Delta \\ 1 \end{bmatrix} \sigma_i \Delta w_{i,k+1}, \tag{7}$$

where the noise term is again only an approximation of the corresponding (more complicated) stochastic integral.

Remark 3. The *nearly constant velocity model*, the *nearly constant acceleration model*, and the *nearly coordinated turn model* (in 2D) are special cases of the above two nearly constant turn models. More precisely, the nearly constant velocity model (or *white-noise acceleration model*) is a special case of (4) with $q_1 = q_2 = q_3 = 0$, and the nearly constant acceleration model (or *Wiener-process acceleration model*) is a special case of (6) with $\omega = 0$. The so-called *coordinated turn model* in 2D is a special case of (2), in which $q_1 = q_2 = 0$ and $q_3 = \omega$:

$$\begin{bmatrix} \dot{v}_1(t) \\ \dot{v}_2(t) \end{bmatrix} = \begin{bmatrix} 0 & -\omega \\ \omega & 0 \end{bmatrix} \begin{bmatrix} v_1(t) \\ v_2(t) \end{bmatrix}.$$

And, as in model (4), if we assume a white noise "relative" acceleration and also include the equations for the position, then we obtain the nearly coordinated turn model or *essentially constant speed model* (see [3][26]):

$$\begin{bmatrix} \dot{x}_1(t) \\ \dot{x}_2(t) \\ \dot{v}_1(t) \\ \dot{v}_2(t) \end{bmatrix} = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & -\omega \\ 0 & 0 & \omega & 0 \end{bmatrix} \begin{bmatrix} x_1(t) \\ x_2(t) \\ v_1(t) \\ v_2(t) \end{bmatrix} + \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} \sigma_1 \dot{w}_1(t) \\ \sigma_2 \dot{w}_2(t) \end{bmatrix}. \tag{8}$$

Its discrete time state equation is obtained as

$$\begin{bmatrix} x_1(t_{k+1}) \\ x_2(t_{k+1}) \\ v_1(t_{k+1}) \\ v_2(t_{k+1}) \end{bmatrix} = \begin{bmatrix} 1 & 0 & \frac{\sin\omega\Delta}{\omega} & -\frac{1-\cos\omega\Delta}{\omega} \\ 0 & 1 & \frac{1-\cos\omega\Delta}{\omega} & \frac{\sin\omega\Delta}{\omega} \\ 0 & 0 & \cos\omega\Delta & -\sin\omega\Delta \\ 0 & 0 & \sin\omega\Delta & \cos\omega\Delta \end{bmatrix} \begin{bmatrix} x_1(t_k) \\ x_2(t_k) \\ v_1(t_k) \\ v_2(t_k) \end{bmatrix} + \begin{bmatrix} \frac{1}{2}\Delta & 0 \\ 0 & \frac{1}{2}\Delta \\ 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} \sigma_1 \Delta w_{1,k+1} \\ \sigma_2 \Delta w_{2,k+1} \end{bmatrix}. \tag{9}$$

2.2 Estimation of Target Orientation

The purpose is to estimate the orientation angles of the target. This is achieved by using the basic models derived above and the Euler's relation between the angular velocity and the orientation angles.

One approach using the first nearly constant turn model has been developed in earlier work of this project (and in particular in Phase-I work of this project). Here we present another approach using the second nearly constant turn model.

To estimate the turn rate, we assume ω is a diffusion process satisfying

$$\dot{\omega}(t) = \alpha_\omega + \gamma_\omega \omega(t) + \sigma_\omega \dot{w}_\omega(t).$$

Then, from this and (6), we obtain a 3-D system

$$\begin{aligned} \dot{v}(t) &= a(t), \\ \dot{a}(t) &= -\omega^2(t) v(t) + \sigma_a \dot{w}_a(t), \\ \dot{\omega}(t) &= \alpha_\omega + \gamma_\omega \omega(t) + \sigma_\omega \dot{w}_\omega(t), \end{aligned} \tag{10}$$

where $\alpha_\omega, \gamma_\omega, \sigma_\omega$, and σ_a are constants, and $w_a(t)$ and $w_\omega(t)$ are independent standard Wiener processes.

As for the measurements, assume the position $x(t)$ of the target is observable at discrete time $t = t_k$. Then $x(t_{k+1}) - x(t_k)$ is also observable. From (7), we have the following observation equation:

$$z(k) = \frac{1}{\omega(t_k)} v(t_k) \sin(\omega(t_k)\Delta) + \frac{1}{\omega^2(t_k)} a(t_k)(1 - \cos(\omega(t_k)\Delta)) + \varepsilon_z v_z(k) , \qquad (11)$$

where $\{v_z(k)\}$ is a sequence of independent Gaussian random variables of zero mean and unit variance, and ε_z is the standard deviation of the noises.

Now we have established model (10)-(11) for the velocity, acceleration and turn rate of the target. To obtain a satisfactory estimate, we need a good filtering algorithm. This will be discussed in Section 3.

From the velocity, acceleration, and turn rate, the angular velocity can be calculated. After obtaining the angular velocity, we can calculate the target orientation as follows. Let $\phi(t) = (\phi_1(t), \phi_2(t), \phi_3(t))$ be the Eulerian orientation angles of the target at time t (resolved in the fixed coordinate system). Then the components (q_1, q_2, q_3) of the angular velocity can be expressed in terms of (ϕ_1, ϕ_2, ϕ_3) (see [24]):

$$\begin{aligned} q_1 &= \dot{\phi}_1 \cos\phi_2 \cos\phi_3 - \dot{\phi}_2 \sin\phi_3, \\ q_2 &= \dot{\phi}_1 \cos\phi_2 \sin\phi_3 + \dot{\phi}_2 \cos\phi_3, \\ q_3 &= -\dot{\phi}_1 \sin\phi_2 + \dot{\phi}_3. \end{aligned} \qquad (12)$$

Solving for $(\dot{\phi}_1, \dot{\phi}_2, \dot{\phi}_3)$ gives

$$\begin{aligned} \dot{\phi}_1 &= (q_1 \cos\phi_3 + q_2 \sin\phi_3) \sec\phi_2, \\ \dot{\phi}_2 &= -q_1 \sin\phi_3 + q_2 \cos\phi_3, \\ \dot{\phi}_3 &= (q_1 \cos\phi_3 + q_2 \sin\phi_3) \tan\phi_2 + q_3. \end{aligned} \qquad (13)$$

Thus, to obtain the conditional expectation and conditional covariance of the orientation angles, one can approximate (13) directly. These are deterministic ordinary differential equations, which can be easily solved by using Runge-Kutta method, for example. (Another approach to obtain the expectation and covariance of the angles is to first replace the dynamics equations by an extended system, including the angles, and also rewrite the observation equation in terms of all the new state variables. Then consider a higher-dimensional filtering problem based on the new equations. This will give the estimation of both the angular velocity and the orientation angles at the same time.)

2.3 Target Tracking with Angle-only Measurements

There are military situations in which it is desirable to estimate the position, velocity and perhaps acceleration of a target from measurements of angle but not range. A well-known example is the determination by a submarine of planar position and velocity of a ship from passive sonar measurements, because the submarine commander does not want to reveal his presence by pinging. In air warfare, a fighter defending against a raid may wish to launch a missile against a jammer at unknown range, but should not do so unless the jammer's position and velocity can be estimated. A more recent problem is estimation of target

position, velocity and acceleration in three dimensions from angle measurements only, either with a passive IR receiver or a jammed radar receiver on a missile, in order to utilize optimal guidance.

There appear to have been two main approaches to angle-only tracking: (a) tracking based on the Extended Kalman Filter in Cartesian coordinates (see [1] [9]); (b) reformulation of the problem in terms of modified polar coordinates with subsequent application of EKF ([14]).

Unfortunately neither one of the above is satisfactory. The main limitation on EKF application to angle-only tracking is the nontrivial nonlinearity of the latter. Its mathematical model can be described as follows (for the sake of simplicity we describe here the 2D case and a 3D example will be given in Section 4):

Signal:

$$dx_1(t) = b_1(x_1(t), x_2(t))dt + \sigma_1 dw_1(t), \quad x_1(0) = x_1^0,$$
$$dx_2(t) = b_2(x_1(t), x_2(t))dt + \sigma_2 dw_2(t), \quad x_2(0) = x_2^0;$$

Observation:

$$dy(t) = \arctan(x_2(t)/x_1(t))dt + \sigma dv(t), \quad y(0) = 0,$$

where $w_1(t), w_2(t)$ and $v(t)$ are independent Wiener processes.

The modified polar coordinates (MPC) introduced by Hoelzer et al. [14] are designed to reduce the nonlinear observations to linear. Unfortunately in doing so MPC also transforms the signal process. Unless the latter is of very simple nature, the MPC transform makes the signal system practically intractable.

Much more perspective approach to angle-only tracking is based on optimal nonlinear filtering. The optimal nonlinear filtering theory allows to compute the posterior density function $p(t,x)$ of the signal process $x(t) = (x_1(t), x_2(t))$ given observations $y(s),\ s \le t$. The posterior density function is defined by $p(t,x) = u(t,x)/\int u(t,x)dx$ where the so called unnormalized filtering density $u(t,x)$ is a solution of the Zakai equation

$$du(t,x) = \mathcal{L}u(t,x)dt + h(t)dy(t), \quad u(0,x) = p(0,x),$$

where

$$\mathcal{L}u = \frac{1}{2}\left(\sigma_1^2\frac{\partial^2 u}{\partial x_1^2} + \sigma_2^2\frac{\partial^2 u}{\partial x_2^2}\right) - \left(\frac{\partial(b_1 u)}{\partial x_1} + \frac{\partial(b_2 u)}{\partial x_2}\right).$$

The optimal estimate of the signal $(x_1(t), x_2(t))$ is given by

$$\hat{x}_i(t) = \frac{\int x_i u(t,x_1,x_2)dx_1dx_2}{\int u(t,x_1,x_2)dx_1dx_2}, \quad i = 1,2.$$

For a long period of time practical application of nonlinear filtering has been strained by numerical difficulties related to on-line solution of the Zakai equation. Recently this problem was resolved with the introduction of a spectral approach to nonlinear filtering by B.L. Rozovskii and his co-workers (see [17]). The spectral approach proposed in these works is based on Wiener Chaos expansion. An important feature of this expansion is that it separates observation $(y(s),\ s \le t)$ and parameters $(b_1, b_2, \sigma_1, \sigma_2, \sigma)$ and $h = \arctan(\frac{x_1}{x_2})$ in Zakai's equation. The numerical algorithm for solving the Zakai equation based on the spectral approach splits into two parts: "deterministic" and "stochastic". The time consuming

computation of the deterministic part can be shifted off-line. The stochastic part involving the observation process $y(t)$ is computationally simple and can be performed in real time.

In the course of this project we compared an angle-only tracker based on a spectral algorithm for nonlinear filtering prediction and smoothing with a standard EKF tracker. We considered several practically important cases, in particular: (1) the target exhibits special evasive maneuvers; (2) lack of prior information about position and/or speed of the target. In all cases the performance of the nonlinear angle-only tracker was superior to the EKF tracker.

3 Fast Nonlinear Filters of Linear Complexity

Our objective here is to develop recursive numerical algorithms for computing the optimal filter in which the on-line computation is as simple as possible; in particular, the number of computer operations at each time step should be proportional to the number of grid points where the filtering density is numerically defined. The starting point in the derivation is the equation for the unnormalized filtering density in the general nonlinear model, and the approach is based on the technique known as operator splitting.

Since in most cases the observational measurements are only available at discrete time moments, in this section we consider dynamic systems with discrete observations.

Two algorithms are developed; one is described in subsection 3.2 and the other in subsection 3.3. The computational complexity of both algorithms is indeed $O(N)$ where N is the number of points in the spatial domain. Their approximation errors are also estimated.

3.1 The Nonlinear Filtering Problem

Now consider the dynamical system described by the stochastic differential equation

$$\begin{aligned} &dX_t = b(X_t)dt + \sigma(X_t)dW_t, \quad t > 0, \\ &X_0 \sim \pi_0(x), \end{aligned} \tag{14}$$

and the discrete observations given by

$$z_k = h(X_{t_k}) + \varepsilon(X_{t_k})V_k, \quad k = 1, 2, \cdots, \tag{15}$$

where $\pi_0 : \mathbb{R}^n \to \mathbb{R}$ is the initial density, $b : \mathbb{R}^n \to \mathbb{R}^n$, and $h : \mathbb{R}^n \to \mathbb{R}^m$ are known vector-valued functions, $\sigma : \mathbb{R}^n \to \mathbb{R}^{n\times r}$ and $\varepsilon : \mathbb{R}^n \to \mathbb{R}^{m\times m}$ are matrices with known function entries, $\{W_t\}_{t\geq 0}$ is a standard r-dimensional Brownian motion, $\{V_k\}_{k\geq 1}$ is a standard d_1-dimensional white Gaussian sequence, and $t_k = k\Delta$ $(\Delta > 0)$. X_0, $\{W_t\}$ and $\{V_k\}$ are assumed to be independent, and functions $b, h, \sigma, \varepsilon$ and π_0 are assumed to be smooth enough (satisfying certain regularity conditions, see [19][23]).

Let $f = f(x), x \in \mathbb{R}^n$, be a measurable scalar function such that $\mathbb{E}|f(X_t)|^2 < \infty$ for all $t \geq 0$. Then the filtering problem for (14)-(15) can be stated as follows: find the minimum variance estimate of $f(X_{t_k})$ given the measurements $z_1, \cdots, z_k$. This estimate is called *the optimal filter* and is known to be

$$\hat{f}_k = \mathbb{E}\Big[f(X_{t_k}) \mid z_1, \cdots, z_k\Big].$$

For computational purposes, the optimal filter can be characterized as follows.

Denote by T_t the solution operator for the Fokker-Planck equation corresponding to the state process; in other words, $u(t,x) = T_t\varphi(x)$ is the solution of the equation

$$\begin{aligned} \frac{\partial u(t,x)}{\partial t} &= \frac{1}{2}\sum_{\mu,\nu=1}^{n} \frac{\partial^2}{\partial x_\mu \partial x_\nu}\Big((\sigma(x)\sigma(x)^T)_{\mu\nu}u(t,x)\Big) \\ &\quad - \sum_{\nu=1}^{n} \frac{\partial}{\partial x_\nu}\Big(b_\nu(x)u(t,x)\Big), \quad t>0, \\ u(0,x) &= \varphi(x), \end{aligned} \tag{16}$$

where $(\sigma(x)\sigma(x)^T)_{\mu\nu} = \sum_{i=1}^{m} \sigma_{\mu i}(x)\sigma_{\nu i}(x)$ is the μ-th row and ν-th column entry of $\sigma(x)\sigma(x)^T$, and $b_\nu(x)$ is the ν-th component of $b(x)$.

Next, define the *unnormalized filtering density* $p_k(x)$, for $x \in I\!R^n$ and $k \geq 0$, by

$$\begin{aligned} p_0(x) &= \pi_0(x), \\ p_k(x) &= \alpha_k(x) T_\Delta p_{k-1}(x), \end{aligned} \tag{17}$$

where

$$\alpha_k(x) = \exp\Big\{-\frac{1}{2}(z_k - h(x))^T(\varepsilon(x)\varepsilon(x)^T)^{-1}(z_k - h(x))\Big\}, \quad k = 1,2,\cdots.$$

Then the optimal filter $\hat{f}_k$ can be written as follows [15]:

$$\hat{f}_k = \frac{\int_{I\!R^n} p_k(x) f(x)dx}{\int_{I\!R^n} p_k(x)dx}. \tag{18}$$

3.2 Splitting of Convection and Diffusion Terms

To compute the unnormalized filtering density, a fast Fokker-Planck solver is needed. In previous work related to this project, two methods have been developed: a method based on the spectral separation approach ([18]), and a method based on the finite element approximation ([19][23]). In this subsection we present another method which is based on the operator-splitting technique ([16][20]). Similar ideas have been used in [13] for solving the Zakai equation arising from image filtering.

We first assume that in the noise term of (14), $n = r$ and the covariance matrix σ is diagonal and constant: $\sigma_{\mu\nu}(x) = \delta_{\mu\nu}a_\nu$. Then the Fokker-Planck equation (16) becomes

$$\begin{aligned} \frac{\partial u(t,x)}{\partial t} &= \frac{1}{2}\sum_{\nu=1}^{n} \frac{\partial^2}{\partial x_\nu^2}\Big(a_\nu^2 u(t,x)\Big) - \sum_{\nu=1}^{n}\frac{\partial}{\partial x_\nu}\Big(b_\nu(x)u(t,x)\Big), \quad t>0, \\ u(0,x) &= \varphi(x)\,. \end{aligned}$$

Its solution can be expressed as

$$T_t\varphi(x) = I\!E\Big[\varphi(\xi_t^x)\exp\Big\{-\int_0^t (\nabla\cdot b)(\xi_s^x)ds\Big\}\Big]\,, \tag{19}$$

where ξ_t^x is a stochastic process satisfying

$$d\xi_t^x = -b(\xi_t^x)dt + \operatorname{diag}(a_\nu)dW_t \ ,$$
$$\mathbb{P}[\xi_0^x = x] = 1 \ .$$

To proceed the splitting of convection and diffusion terms, denote by T_t^c and T_t^d the solution operators of the equations

$$\frac{\partial u(t,x)}{\partial t} = -\sum_{\nu=1}^{n} \frac{\partial}{\partial x_\nu}\Big(b_\nu(x)u(t,x)\Big), \quad t>0,$$
$$u(0,x) = \varphi(x) \ ,$$

and

$$\frac{\partial u(t,x)}{\partial t} = \frac{1}{2}\sum_{\nu=1}^{n} \frac{\partial^2}{\partial x_\nu^2}\Big(a_\nu^2 u(t,x)\Big), \quad t>0,$$
$$u(0,x) = \varphi(x) \ ,$$

respectively. In fact, the two solution operators can be expressed explicitly as

$$T_t^c\varphi(x) = \varphi(\eta_t^x)\exp\Big\{-\int_0^t (\nabla\cdot b)(\eta_s^x)ds\Big\} \ , \tag{20}$$

and

$$T_t^d\varphi(x) = \mathbb{E}[\varphi(\zeta_t^x)] = \frac{(\pi t)^{-n/2}}{a_1\cdots a_n}\int_{\mathbb{R}^3} \exp\Big\{-\sum_{\nu=1}^{n}\frac{(x_i-y_i)^2}{2a_\nu^2 t}\Big\}\varphi(y)dy \ , \tag{21}$$

where processes η_t^x (deterministic) and ζ_t^x satisfy

$$d\eta_t^x = -b(\eta_t^x)dt, \quad \eta_0^x = x \ ,$$

and

$$d\zeta_t^x = \operatorname{diag}(a_\nu)dW_t, \quad \zeta_0^x = x \ .$$

Then it can be shown that the following approximation formulas hold:

$$T_\Delta\varphi = T_\Delta^d T_\Delta^c\varphi + O(\Delta) \ , \tag{22}$$
$$T_\Delta\varphi = T_{\frac{\Delta}{2}}^c T_\Delta^d T_{\frac{\Delta}{2}}^c\varphi + O(\Delta^2) \ . \tag{23}$$

Therefore, instead of solving the original Fokker-Planck equation, we only need to compute (20) and (21). To compute (20), we only need to solve an ordinary differential equation. To compute (21), we only need to integrate over a small area near point x, especially when the noises are small. In the case of large noises, the multi-grid method ([12]) can be used to achieve linear computational complexity.

Remark 1. The Trotter's semigroup approximation theorem guarantees the convergence of the above approximations (22) and (23) (but it does not provide any error estimate). In our notation it implies

$$T_t\varphi = \lim_{k\to\infty}(T_{t/k}^d T_{t/k}^c)^k\varphi \ .$$

Remark 2. To see the difference between the exact solution and the approximate solution (by splitting), we note that

$$T_t^d T_t^c \varphi(x) = \mathbb{E}\Big[T_t^c \varphi(\zeta_t^x)\Big] = \mathbb{E}\Big[\varphi\left(\eta_t(\zeta_t(x))\right) \exp\Big\{-\int_0^t (\nabla \cdot b)(\eta_s(\zeta_t(x)))ds\Big\}\Big],$$

$$T_t^c T_t^d T_t^c \varphi(x) = T_t^d T_t^c \varphi(\eta_{\frac{t}{2}}^x) \exp\Big\{-\int_0^{t/2} (\nabla \cdot b)(\eta_s^x) ds\Big\}$$

$$= \mathbb{E}\Big[\varphi\left(\eta_{\frac{t}{2}}(\zeta_t(\eta_{\frac{t}{2}}(x)))\right) \exp\Big\{-\int_0^{\frac{t}{2}} (\nabla \cdot b)(\eta_s(\zeta_t(\eta_{\frac{t}{2}}(x))))ds\Big\}\Big] \exp\Big\{-\int_0^{\frac{t}{2}} (\nabla \cdot b)(\eta_s(x))ds\Big\}$$

$$= \mathbb{E}\Big[\varphi\left(\eta_{\frac{t}{2}}(\zeta_t(\eta_{\frac{t}{2}}(x)))\right) \exp\Big\{-\int_0^{\frac{t}{2}} (\nabla \cdot b)\eta_s(x))ds - \int_{\frac{t}{2}}^t (\nabla \cdot b)\eta_s(\zeta_t(\eta_{\frac{t}{2}}(x)), \frac{t}{2})ds\Big\}\Big],$$

where for obvious reasons we have written $\eta_t^x = \eta_t(x)$, $\zeta_t^x = \zeta_t(x)$, and $\eta_t^{x,t'} = \zeta_t(x, t')$, the last being the process η_t starting from the point x at time t'. Comparing these results with (19), we find that in effect our approximation is to split the process ξ_t into two simpler processes: a deterministic process η_t and (up to a constant) a Wiener process ζ_t.

In general, if the covariance matrix $\sigma = \sigma(x)$ is not constant (but still assumed to be diagonal to simplify expressions, i.e., $\sigma_{\mu\nu}(x) = \delta_{\mu\nu} a_\nu(x)$), then we rewrite the Fokker-Planck equation in the following form:

$$\begin{aligned}\frac{\partial u(t,x)}{\partial t} &= \frac{1}{2} \sum_{\nu=1}^n \frac{\partial}{\partial x_\nu} \Big(a_\nu^2(x) \frac{\partial}{\partial x_\nu} u(t,x)\Big) \\ &\quad + \sum_{\nu=1}^n \frac{\partial}{\partial x_\nu} \Big(\Big(a_\nu(x) \frac{\partial}{\partial x_\nu} a_\nu(x) - b_\nu(x)\Big) u(t,x)\Big),\end{aligned}$$

and then split it into the following two equations:

$$\frac{\partial u(t,x)}{\partial t} = \sum_{\nu=1}^n \frac{\partial}{\partial x_\nu} \Big(\Big(a_\nu(x) \frac{\partial}{\partial x_\nu} a_\nu(x) - b_\nu(x)\Big) u(t,x)\Big),$$

and

$$\frac{\partial u(t,x)}{\partial t} = \frac{1}{2} \sum_{\nu=1}^n \frac{\partial}{\partial x_\nu} \Big(a_\nu^2(x) \frac{\partial}{\partial x_\nu} u(t,x)\Big),$$

both of which can be solved in linear computational complexity.

In this general situation, the three solution operators T_t, T_t^c, and T_t^d can also be expressed via certain stochastic (or even deterministic) processes. Indeed, similar to formulas (19), (20), and (21), we have

$$T_t \varphi(x) = \mathbb{E}\Big[\varphi(\xi_t^x) \exp\Big\{\int_0^t \nabla \cdot (\tilde{a} - b)(\xi_s^x) ds\Big\}\Big],$$

$$T_t^c \varphi(x) = \varphi(\eta_t^x) \exp\Big\{\int_0^t \nabla \cdot (\tilde{a} - b)(\eta_s^x) ds\Big\},$$

and

$$T_t^d \varphi(x) = \mathbb{E}[\varphi(\zeta_t^x)]\ ,$$

where processes ξ_t^x, η_t^x, and ζ_t^x satisfy

$$d\xi_t^x = (2\tilde{a}(\xi_t^x) - b(\xi_t^x))dt + \operatorname{diag}(a_\nu(\xi_t^x))dW_t, \quad \xi_0^x = x,$$

$$d\eta_t^x = (\tilde{a}(\eta_t^x) - b(\eta_t^x))dt, \quad \eta_0^x = x,$$

and

$$d\zeta_t^x = \tilde{a}(\zeta_t^x)dt + \operatorname{diag}(a_\nu(\zeta_t^x))dW_t, \quad \zeta_0^x = x,$$

respectively. Here we have denoted by $\tilde{a}(y)$ the vector with components $a_\nu(y)\frac{\partial}{\partial y_\nu}a_\nu(y)$ ($\nu = 1, \cdots, n$).

Remark 3. The algorithm by splitting the convection (or drift) term from the (pure) diffusion term as discussed above has a further advantage that much of the computation in (20) and (21) or their generalizations can be performed before the observations are available. This pre-calculation can save the on-line cost and further speed up the real time performance.

3.3 New Alternating Direction Implicit Schemes

In the previous subsection we used the operator-splitting technique to split the whole process into a convection process and a diffusion process (without "drift"). In this subsection we will apply the operator-splitting technique to split a multi-dimensional process into several one-dimensional processes, since the filtering densities for 1-D processes can be computed easily (with linear computational complexity). And this is the idea of alternating direction implicit (ADI) schemes ([11][16][20][27]).

Here we present two new splitting schemes based on the discretization method described in [25]; one is similar to the Dyakonov scheme ([7][8]) and the other is similar to the Peaceman-Rachford scheme ([22]). We first discuss the 2D case in details and then generalize the results to higher dimensions. Our higher dimensional generalization of the 2D Peaceman-Rachford-like scheme is much simpler than the usual ADI modifications of Peaceman-Rachford scheme in higher dimensions ([5][6][16][20]).

First, let us consider the two-dimensional convection-diffusion equation

$$u_t = au_{xx} + bu_{yy} + cu_x + du_y + eu, \quad \text{in } (0,T] \times \Omega, \tag{24}$$

and the initial-boundary value conditions

$$\begin{aligned} u(0,x,y) &= u^0(x,y), \quad (x,y) \in \Omega, \\ u(t,x,y) &= 0, \quad t \in (0,T], (x,y) \in \partial\Omega, \end{aligned} \tag{25}$$

where $\Omega = (\alpha,\beta) \times (\gamma,\delta), \alpha < \beta, \gamma < \delta, T > 0$; a, b, c, d, and e are known smooth functions with bounded derivatives, defined in $\Omega_T := [0,T] \times \bar{\Omega}$; $u^0 \in C(\bar{\Omega})$ satisfies the continuity condition $u^0(x,y) = 0, \forall (x,y) \in \partial\Omega$. For simplicity, we assume $\min(a,b) \geq \lambda > 0$ (λ a constant), and $e \leq 0$.

Let $x_i := \alpha + i\Delta x$ ($0 \leq i \leq N_x$), $y_j := \gamma + j\Delta y$ ($0 \leq j \leq N_y$), and $t_k := k\Delta t$ ($0 \leq k \leq M$) be a partition of Ω_T, with $\Delta x := \frac{\beta-\alpha}{N}, \Delta y := \frac{\delta-\gamma}{N}$, and $\Delta t := \frac{T}{M}$. For any $v \in C(\Omega_T)$, denote $v_{ij}^k := v(t_k, x_i, y_j)$ ($0 \leq i \leq N_x, 0 \leq j \leq N_y, 0 \leq k \leq M$).

We approximate problem (24)-(25) by the following difference equations:

$$
\begin{aligned}
\frac{D^+ v_{ij}^k}{\Delta t} &= \left(a_{ij}^{k+\frac{1}{2}}\frac{D_{+x}D_{-x}}{\Delta x^2} + b_{ij}^{k+\frac{1}{2}}\frac{D_{+y}D_{-y}}{\Delta y^2}\right)\left(\frac{v_{ij}^{k+1}+v_{ij}^k}{2}\right) \\
&\quad + c_{ij,+}^{k+\frac{1}{2}}\left(\frac{D_{+x}v_{ij}^{k+1}+D_{-x}v_{ij}^k}{2\Delta x}\right) + c_{ij,-}^{k+\frac{1}{2}}\left(\frac{D_{-x}v_{ij}^{k+1}+D_{+x}v_{ij}^k}{2\Delta x}\right) \\
&\quad + d_{ij,+}^{k+\frac{1}{2}}\left(\frac{D_{+y}v_{ij}^{k+1}+D_{-y}v_{ij}^k}{2\Delta y}\right) + d_{ij,-}^{k+\frac{1}{2}}\left(\frac{D_{-y}v_{ij}^{k+1}+D_{+y}v_{ij}^k}{2\Delta y}\right) \\
&\quad + e_{ij}^{k+\frac{1}{2}}\left(\frac{v_{ij}^{k+1}+v_{ij}^k}{2}\right), \\
&\quad i=1,\cdots,N_x-1,\ j=1,\cdots,N_y-1,\ k=0,1,\cdots,M-1; \qquad (26)\\
v_{ij}^0 &= u^0(x_i,y_j), \quad i=0,1,\cdots,N_x,\ j=0,1,\cdots,N_y, \\
v_{0,j}^k &= 0,\ \ v_{N_x,j}^k=0, \quad j=1,\cdots,N_y-1,\ k=1,\cdots,M, \\
v_{i,0}^k &= 0,\ \ v_{i,N_y}^k=0, \quad i=1,\cdots,N_x-1,\ k=1,\cdots,M, \qquad (27)
\end{aligned}
$$

where $D_{\pm x}$ and $D_{\pm y}$ denote the forward and backward difference operators in the x and y directions, respectively, $c_{ij,\pm}^{k+\frac{1}{2}}$ are defined as

$$
\begin{aligned}
c_{ij,+}^{k+\frac{1}{2}} &:= c_{ij}^{k+\frac{1}{2}}\mathbb{1}_{[c_{ij}^{k+\frac{1}{2}}\geq 0]} = \max(0,c_{ij}^{k+\frac{1}{2}}) = \frac{c_{ij}^{k+\frac{1}{2}}+|c_{ij}^{k+\frac{1}{2}}|}{2}, \\
c_{ij,-}^{k+\frac{1}{2}} &:= c_{ij}^{k+\frac{1}{2}}\mathbb{1}_{[c_{ij}^{k+\frac{1}{2}}< 0]} = \min(0,c_{ij}^{k+\frac{1}{2}}) = \frac{c_{ij}^{k+\frac{1}{2}}-|c_{ij}^{k+\frac{1}{2}}|}{2},
\end{aligned}
$$

and similarly for $d_{ij,\pm}^{k+\frac{1}{2}}$. It can be shown (see [25]) that the truncation error of scheme (26) is $O\Big((\Delta t+\Delta x+\Delta y)^2\Big)$ and that the scheme is unconditionally stable.

Multiplying both sides of (26) by Δt, we can rewrite system (26)-(27) into the following matrix-vector form:

$$
\left(I-\frac{\Delta t}{2}(A_{x,1}+A_{y,1})\right)v^{k+1} = \left(I+\frac{\Delta t}{2}(A_{x,0}+A_{y,0})\right)v^k, \tag{28}
$$

where matrices $A_{x,0}, A_{x,1}, A_{y,0}, A_{y,1}$ are "tridiagonal": all but two of their off-diagonals are zero and the two nonzero off-diagonals are of the same distance from the diagonal; and each of these matrices contains $e_{ij}^{k+\frac{1}{2}}/2$ in the diagonal entries. (In general, these coefficient matrices depend on the time step k, but we have ommitted the superscript $k+\frac{1}{2}$ to simplify the discussions that follow.)

As usual, equation (28) is numerically much more difficult than its one-dimensional analogue (see [25]). As a resolution to overcome this difficulty, we replace (28) by the following "one-dimensionalized" approximation

$$
\left(I-\frac{\Delta t}{2}A_{x,1}\right)\left(I-\frac{\Delta t}{2}A_{y,1}\right)v^{k+1} = \left(I+\frac{\Delta t}{2}A_{x,0}\right)\left(I+\frac{\Delta t}{2}A_{y,0}\right)v^k. \tag{29}
$$

Now we only need to solve a sequence of tridiagonal systems at each time step.

Since scheme (29) differs from scheme (28) only by the term

$$\frac{\Delta t^2}{4}\left(A_{x,1}A_{y,1}v^{k+1} - A_{x,0}A_{y,0}v^k\right) ,$$

which is easily checked to be of order $O(\Delta t^2(\Delta t + \Delta x + \Delta y))$ (meaning a difference of truncation error by $O(\Delta t^2 + \Delta t\Delta x + \Delta t\Delta y)$), the accuracy of scheme (29) remains the same order, that is, $O((\Delta t + \Delta x + \Delta y)^2)$.

An alternative approach is to use the following alternating direction implicit (ADI) scheme:

$$\begin{aligned}\left(I - \frac{\Delta t}{2}A_{x,1}\right) v^{k+\frac{1}{2}} &= \left(I + \frac{\Delta t}{2}A_{y,0}\right) v^k ,\\ \left(I - \frac{\Delta t}{2}A_{y,1}\right) v^{k+1} &= \left(I + \frac{\Delta t}{2}A_{x,0}\right) v^{k+\frac{1}{2}} .\end{aligned} \tag{30}$$

To obtain the truncation error of this Peaceman-Rachford-like approximation, we notice that equations (29) and (30) can be written as follows:

$$v^{k+1} = \left(I - \frac{\Delta t}{2}A_{y,1}\right)^{-1}\left(I - \frac{\Delta t}{2}A_{x,1}\right)^{-1}\left(I + \frac{\Delta t}{2}A_{x,0}\right)\left(I + \frac{\Delta t}{2}A_{y,0}\right) v^k; \tag{31}$$

$$v^{k+1} = \left(I - \frac{\Delta t}{2}A_{y,1}\right)^{-1}\left(I + \frac{\Delta t}{2}A_{x,0}\right)\left(I - \frac{\Delta t}{2}A_{x,1}\right)^{-1}\left(I + \frac{\Delta t}{2}A_{y,0}\right) v^k. \tag{32}$$

When compared with (31), the term $(I - \frac{\Delta t}{2}A_{x,1})^{-1}(I + \frac{\Delta t}{2}A_{x,0})$ is replaced by $(I + \frac{\Delta t}{2}A_{x,0})(I - \frac{\Delta t}{2}A_{x,1})^{-1}$ in (32). But both of these are solving the same "one-dimensinal" convection-diffusion equation with the same order of approximation error. So the accuracy of scheme (32) is the same as that of (31), i.e., $O((\Delta t + \Delta x + \Delta y)^2)$.

Note that if traditional Crank-Nicholson discretization is used, i.e., if $A_{x,1} = A_{x,0}$ and $A_{y,1} = A_{y,0}$, then (29) and (30) become Dyakonov and Peaceman-Rachford schemes, respectively. In this case, the two schemes are in fact equivalent; i.e., in two dimensions, Dyakonov scheme and Peaceman-Rachford scheme are just two different forms of the same approximation. On the other hand, with our discretization, schemes (29) and (30) generally are not equivalent.

The above two-dimensional schemes can be generalized to higher dimensions. Indeed, similar to (31) and (32), we may use one of the following two splitting schemes for three-dimensional homogeneous problems:

$$\begin{aligned} v^{k+1} = {} & \left(I - \frac{\Delta t}{2}A_{x,1}\right)^{-1}\left(I - \frac{\Delta t}{2}A_{y,1}\right)^{-1}\left(I - \frac{\Delta t}{2}A_{z,1}\right)^{-1} \\ & \left(I + \frac{\Delta t}{2}A_{z,0}\right)\left(I + \frac{\Delta t}{2}A_{y,0}\right)\left(I + \frac{\Delta t}{2}A_{x,0}\right) v^k ; \end{aligned} \tag{33}$$

$$\begin{aligned} v^{k+1} = {} & \left(I - \frac{\Delta t}{2}A_{x,1}\right)^{-1}\left(I + \frac{\Delta t}{2}A_{y,0}\right)\left(I - \frac{\Delta t}{2}A_{z,1}\right)^{-1} \\ & \left(I + \frac{\Delta t}{2}A_{z,0}\right)\left(I - \frac{\Delta t}{2}A_{y,1}\right)^{-1}\left(I + \frac{\Delta t}{2}A_{x,0}\right) v^k . \end{aligned} \tag{34}$$

And in the four-dimensional case, we use one of the following schemes:

$$
\begin{aligned}
v^{k+1} &= \left(I-\frac{\Delta t}{2}A_{x_1,1}\right)^{-1}\left(I-\frac{\Delta t}{2}A_{x_2,1}\right)^{-1}\left(I-\frac{\Delta t}{2}A_{x_3,1}\right)^{-1}\left(I-\frac{\Delta t}{2}A_{x_4,1}\right)^{-1} \\
&\quad \left(I+\frac{\Delta t}{2}A_{x_4,0}\right)\left(I+\frac{\Delta t}{2}A_{x_3,0}\right)\left(I+\frac{\Delta t}{2}A_{x_2,0}\right)\left(I+\frac{\Delta t}{2}A_{x_1,0}\right)v^k\,; \qquad (35) \\
v^{k+1} &= \left(I-\frac{\Delta t}{2}A_{x_1,1}\right)^{-1}\left(I+\frac{\Delta t}{2}A_{x_2,0}\right)\left(I-\frac{\Delta t}{2}A_{x_3,1}\right)^{-1}\left(I+\frac{\Delta t}{2}A_{x_4,0}\right) \\
&\quad \left(I-\frac{\Delta t}{2}A_{x_4,1}\right)^{-1}\left(I+\frac{\Delta t}{2}A_{x_3,0}\right)\left(I-\frac{\Delta t}{2}A_{x_2,1}\right)^{-1}\left(I+\frac{\Delta t}{2}A_{x_1,0}\right)v^k. \qquad (36)
\end{aligned}
$$

The same arguments as in the 2D case can be used to show that all these schemes are second order accurate. And it can also be shown that all these schemes are absolutely stable.

Finally, we make two remarks about the above ADI schemes.

1. Whereas the order of the alternating directions or variables in the Dyakonov-like schemes (31), (33) and (35) does not affect the second-order accuracy (up to higher order terms as long as the explicit and implicit schemes are applied separately), an impropriate order of alternating directions in the Peaceman-Rachford-like schemes (32), (34) and (36) would lead to first-order accuracy. For example, in the 3D case, in (33) we may also use the order y, z, x, z, y, x instead of x, y, z, z, y, x. But in (34), we can only change the order to something like y, z, x, x, z, y; the second half must be in the inverse order of the first half.

2. Our higher dimensional generalizations (34) and (36) are simpler than the usual higher dimensional ADI modifications of Peaceman-Rachford scheme, even for Crank-Nicholson discretization with $A_{x_\nu,0} = A_{x_\nu,1} = A_{x_\nu}, \forall \nu$.

4 Numerical Example

To illustrate the performance of the above algorithm, let us consider the following three-dimensional tracking problem with angle-only observations

$$
\begin{aligned}
\text{Signal:}\quad & \dot{X}_1(t) = b_{11}X_2^3(t) + b_{12}X_3(t) + \sigma_{11}\dot{W}_1(t), \\
& \dot{X}_2(t) = b_2 + \sigma_{22}\dot{W}_2(t), \\
& \dot{X}_3(t) = b_3 + \sigma_{33}\dot{W}_3(t), \\
\text{Observation:}\quad & z_1(k) = \operatorname{sign}(X_2(t_k)) \arccos \frac{X_1(t_k)}{\sqrt{X_1^2(t_k)+X_2^2(t_k)}} + v_1(k), \\
& z_2(k) = \arcsin \frac{X_3(t_k)}{\sqrt{X_1^2(t_k)+X_2^2(t_k)+X_3^2(t_k)}} + v_2(k),
\end{aligned}
$$

where $b_{11} = -200$, $b_{12} = 50$, $b_2 = -1/2$, $b_2 = -1/4$, $\sigma_{11} = 0.045$, $\sigma_{22} = 0.023$, $\sigma_{33} = 0.012$, $t_k = k\Delta$, $\Delta = 0.01$, $v_1(k) \sim N(0, .64^2)$, $v_2(k) \sim N(0, .36^2)$, and the initial target state $(X_1(0), X_2(0), X_3(0))$ has joint density $\pi_0(x_1, x_2, x_3) = \frac{1}{c_0}\exp(-100(x_1-0.45)^2 - 121(x_2 - 0.42)^2 - 64(x_3-0.20)^2)$, c_0 being the normalizing constant.

In our experiments, we took $T = 2.0$ (so $M = 200$), $S = [-0.95, 0.75] \times [-0.60, 0.60] \times [-0.30, 0.30]$, $x_1^i = -0.95 + i\Delta x_1$ $(0 \le i \le n_1)$, $x_2^j = -0.60 + j\Delta x_2$ $(0 \le j \le n_2)$, $x_3^k = -0.30 + k\Delta x_3$ $(0 \le k \le n_3)$, $\Delta x_1 = \frac{1.6}{n_1}$, $\Delta x_2 = \frac{1.2}{n_2}$, $\Delta x_3 = \frac{0.6}{n_3}$, and $[n_1, n_2, n_3] = [40, 30, 20]$ or $[80, 60, 30]$.

Results of typical tracking steps are shown in the figures at the end of this report, where the true initial target position was $(X_1(0), X_2(0), X_3(0)) = (0.45, 0.5, 0.25)$. Programming source code for this and other examples is also attached.

In conclusion, the filtering algorithms proposed in this report are based on the exact optimal filter and so apply to systems with high nonlinearity and different noise levels. On the other hand, the new algorithms are fast, in that their calculations have linear comlexity per time step.

References

[1] V. J. Aidala, Kalman filter behavior in bearings-only tracking applications, *IEEE Trans. on Aerospace and Electronic Systems*, Vol. AES-15, No.1, January 1979, pp. 29-39.

[2] V. I. Arnold, *Mathematical Methods of Classical Mechanics, 2nd Ed.* New York: Springer-Verlag, 1989.

[3] Y. Bar-Shalom and X.-R. Li, *Estimation and Tracking: Principles, Techniques, and Software.* Boston: Artech House, Inc., 1993.

[4] W. D. Blair and G. A. Watson, IMM algorithm for solution to benchmark problem for tracking maneuvering targets. *Proceedings of SPIE Acquisition, Tracking, and Pointing VIII* (Orlando, FL.), SPIE, 1994, pp. 476-488.

[5] J. Douglas and J. Gunn. A general formulation of alternating direction methods I: parabolic and hyperbolic problems. *Numer. Math.*, Vol. 6 (1964), pp. 428-453.

[6] J. Douglas and H. H. Rachford. On the numerical solution of the heat conduction problems in two and three space variables. *Trans. Amer. Math. Soc.*, Vol. 82 (1956), pp. 421-439.

[7] E. G. Dyakonov. Difference schemes with split operators for unsteady equations. *Dokl. Akad. Nauk SSSR*, Vol. 144 (1962), pp. 29-32.

[8] E. G. Dyakonov. Difference schemes with split operators for general parabolic equations of second order with variable coefficients. *Zh. Vychisl. Mat. i Mat. Fiz.*, Vol. 4 (1964), pp. 278-291. English translation: *U.S.S.R. Comp. Math.*, Vol. 4 (1964), pp. 92-110.

[9] F. El-Hawary and Y. Jing, Robust regression-based EKF for tracking underwater targets, *IEEE J. of Oceanic Engineering*, Vol. 20, No.1 1995, pp. 31-41.

[10] B. Etkin. *Dynamics of Atmospheric Flight.* New York: John Wiley & Sons, Inc., 1972.

[11] B. Gustafsson, H.-O. Kreiss, and J. Oliger. *Time Dependent Problems and Difference Methods.* John Wiley & Sons, Inc., New York, 1995.

[12] W. Hackbusch. *Multi-grid Methods and Applications.* Springer-Verlag, Berlin, 1985.

[13] Z. S. Haddard and S. R. Simanca. Filtering image records using wavelets and the Zakai equation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Vol. 17, No. 11, Nov. 1995, pp. 1069-1078.

[14] H. D. Hoelzer, G. W. Jonson, and A. O. Cohen, Modified Polar Coordinates – The Key to Well Behaved Bearings Only Ranging, IR&D Report 78-M19-0001A, IBM Federal Systems Division, Shipboard and Defense Systems, Manassas VA 22110, August 31, 1978.

[15] A. H. Jazwinski. *Stochastic Processes and Filtering Theory.* Academic Press, New York, 1970.

[16] L. Kang et al. *Decomposition Methods for Numerically Solving Higher Dimensional Partial Differential Equations.* Shanghai Science and Technology Press, Shanghai, 1990.

[17] S. V. Lototsky, R. Mikulevicius, B. L. Rozovksii. Nonlinear filtering revisited: a spectral approach. *SIAM J. Control Optim.*, Vol. 35, No.2, March 1997, pp. 435-461.

[18] S. V. Lototsky and B. L. Rozovskii. Recursive nonlinear filter for a continuous-discrete time model: separation of parameters and observations. *IEEE Transactions on Automatic Control*, 1997. (To appear)

[19] S. V. Lototsky, C. Rao, B. L. Rozovskii. Fast nonlinear filter for continuous-discrete time multiple models. *Proceedings of the 35th Conference on Decision and Control*, Kobe, Japan, 1996, Omnipress, Madison, Wisconsin, Vol. 4, pp. 4060-4064.

[20] G. I. Marchuk. Splitting and alternating direction methods. In: *Handbook of Numerical Analysis, Vol. I*, Ph. G. Ciarlet and J.-L. Lions (eds.), North-Holland, Amsterdam, 1990, pp. 197-462.

[21] G. N. Milstein, *Numerical Integration of Stochastic Differential Equations.* Dordrecht: Kluwer Academic Publishers, 1995.

[22] D. W. Peaceman and H. H. Rachford Jr. The numerical solution of parabolic and elliptic differential equations. *J. SIAM*, 3 (1955), 28-41.

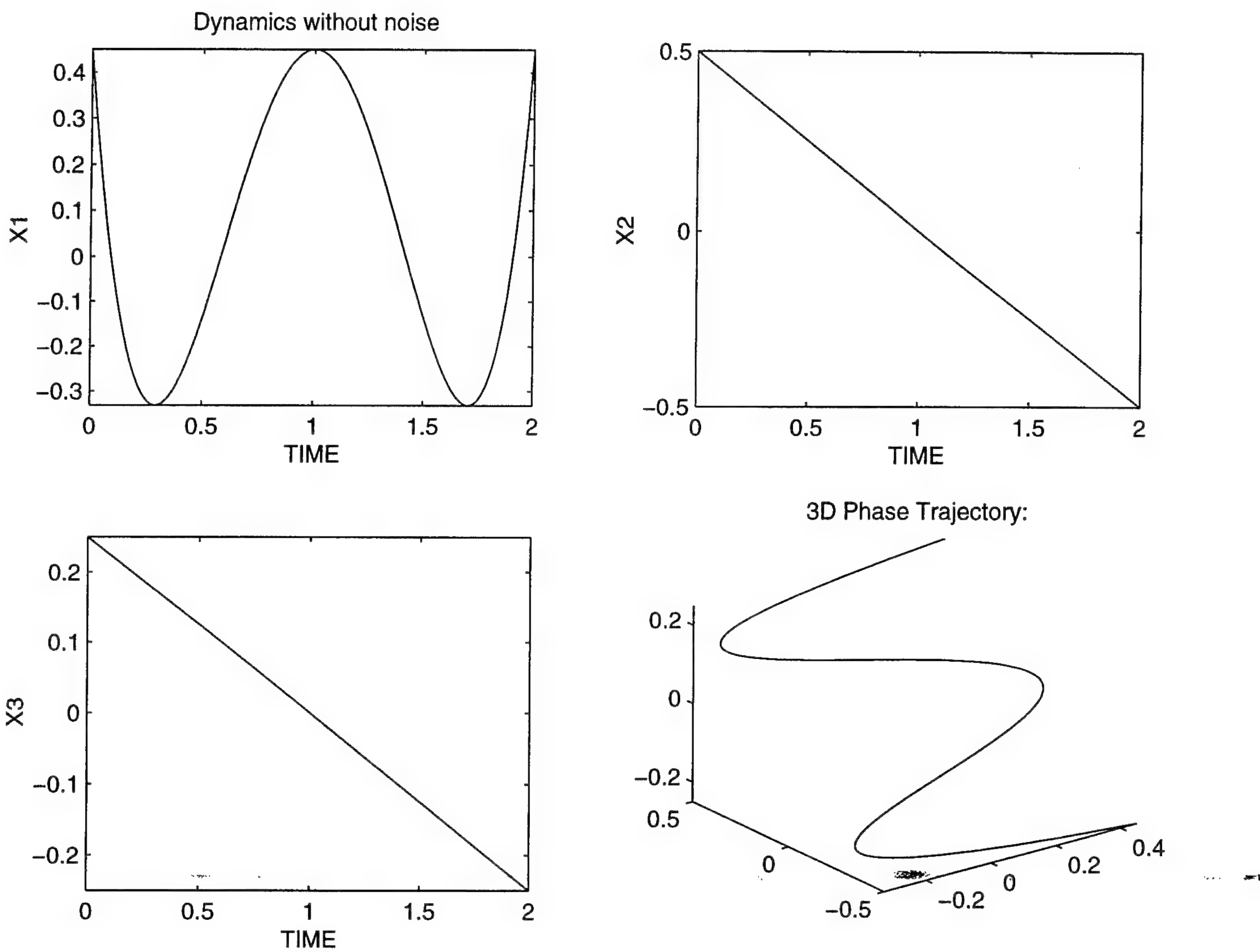
[23] C. Rao and B. L. Rozovskii. A fast filter for nonlinear systems with discrete observations. *Proceedings of the Second IASC World Conference*, Pasadena, February 1997.

[24] C. Rao. Mathematical models of maneuvering targets. Technical Report, University of Southern California, 1996.

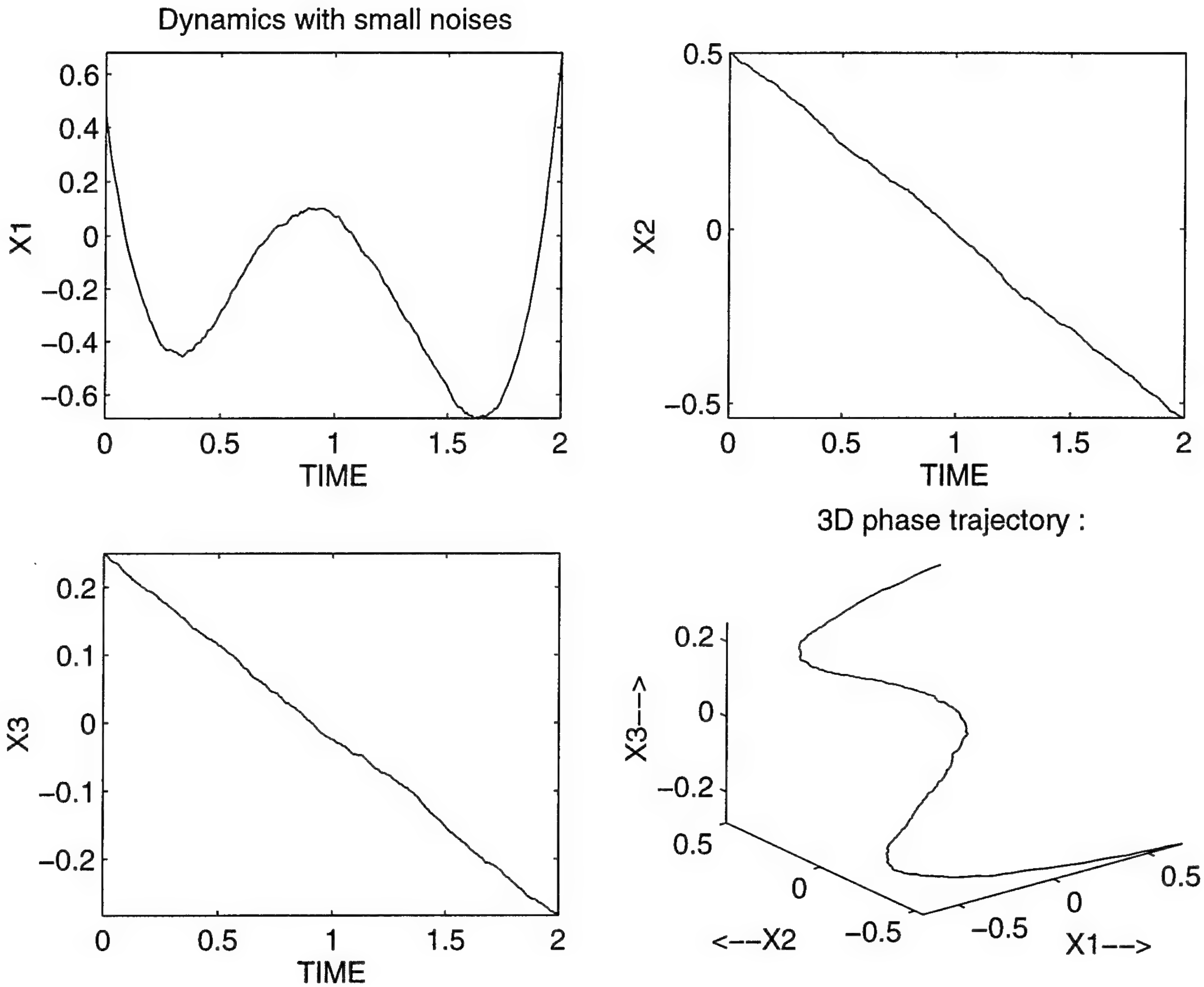
[25] C. Rao. A new difference scheme and related splitting methods for convection-diffusion equations. Technical Report, University of Southern California, 1997.

[26] D. D. Sworder, P. F. Singer, D. Doria, and R. G. Hutchins, Image-enhanced estimation methods. *Proceedings of the IEEE*, Vol. 81 (1993), No. 6, pp. 797-812.

[27] N. N. Yanenko. *The Method of Fractional Steps: The Solution of Problems of Mathematical Physics in Several Variables.* Springer-Verlag, Berlin, 1971.

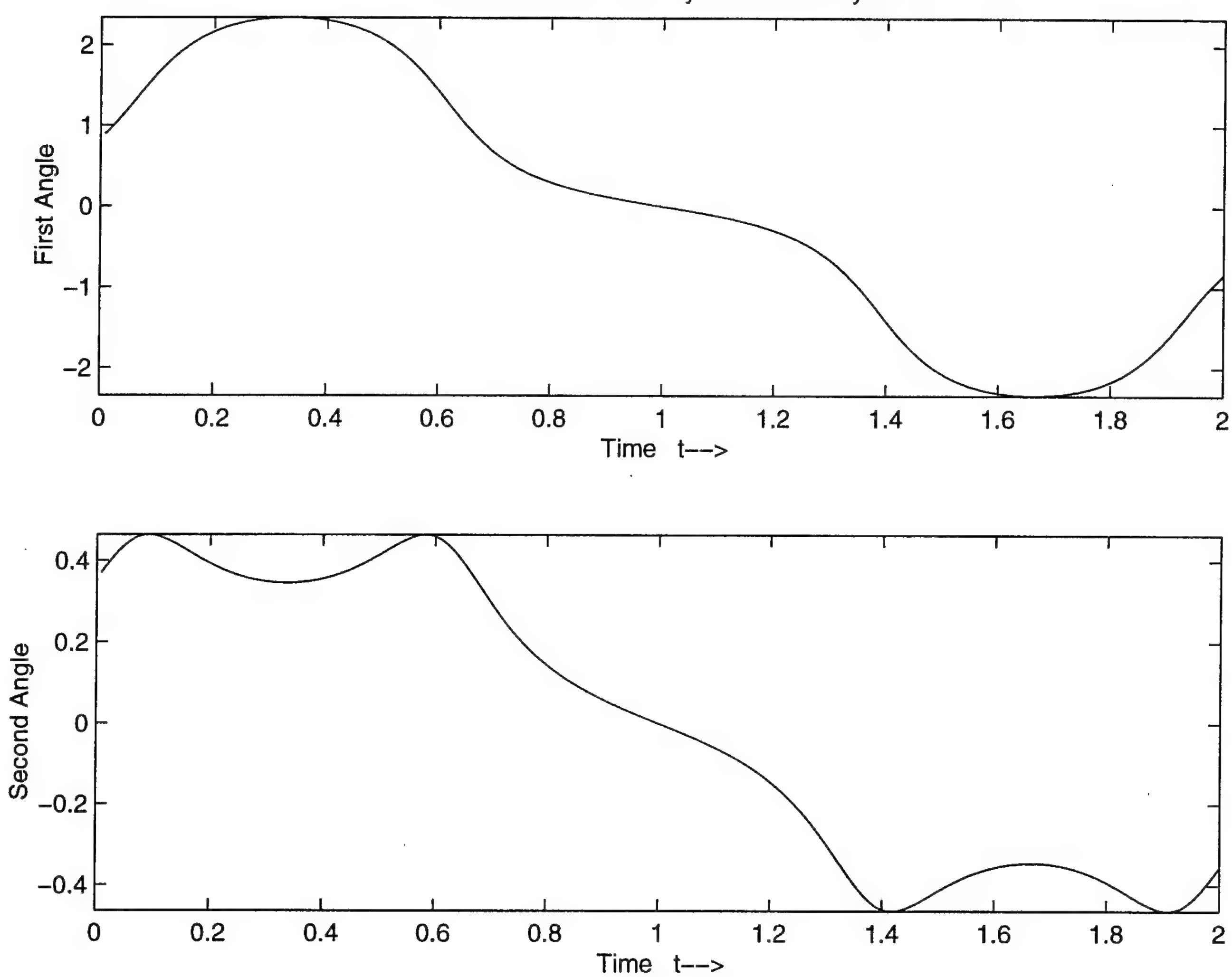


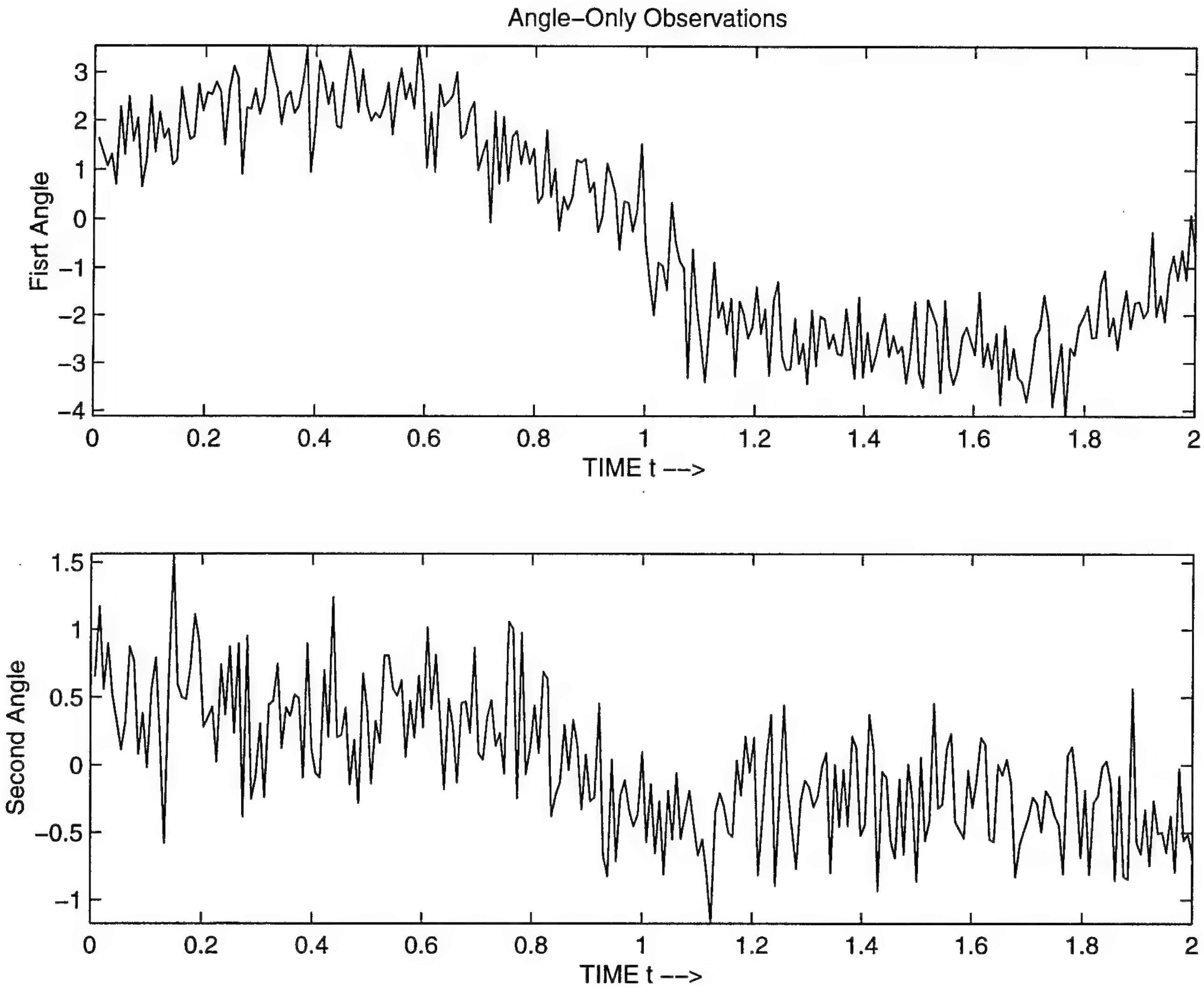
Dynamics without noise
X1
TIME
X2
TIME
X3
TIME
3D Phase Trajectory:

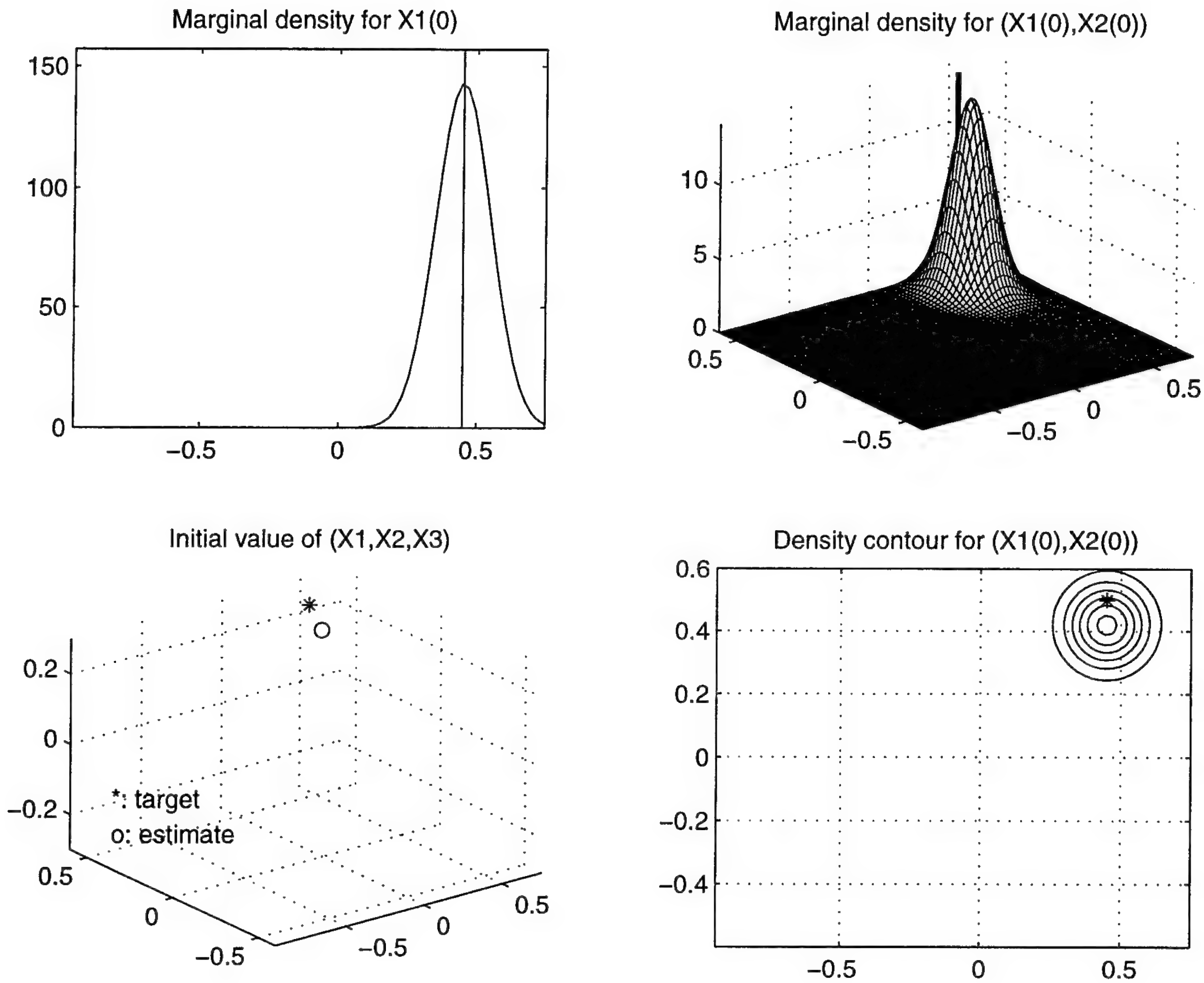


Dynamics with small noises
X1
TIME
X2
TIME
X3
TIME
3D phase trajectory :
X3--->
<---X2
X1--->

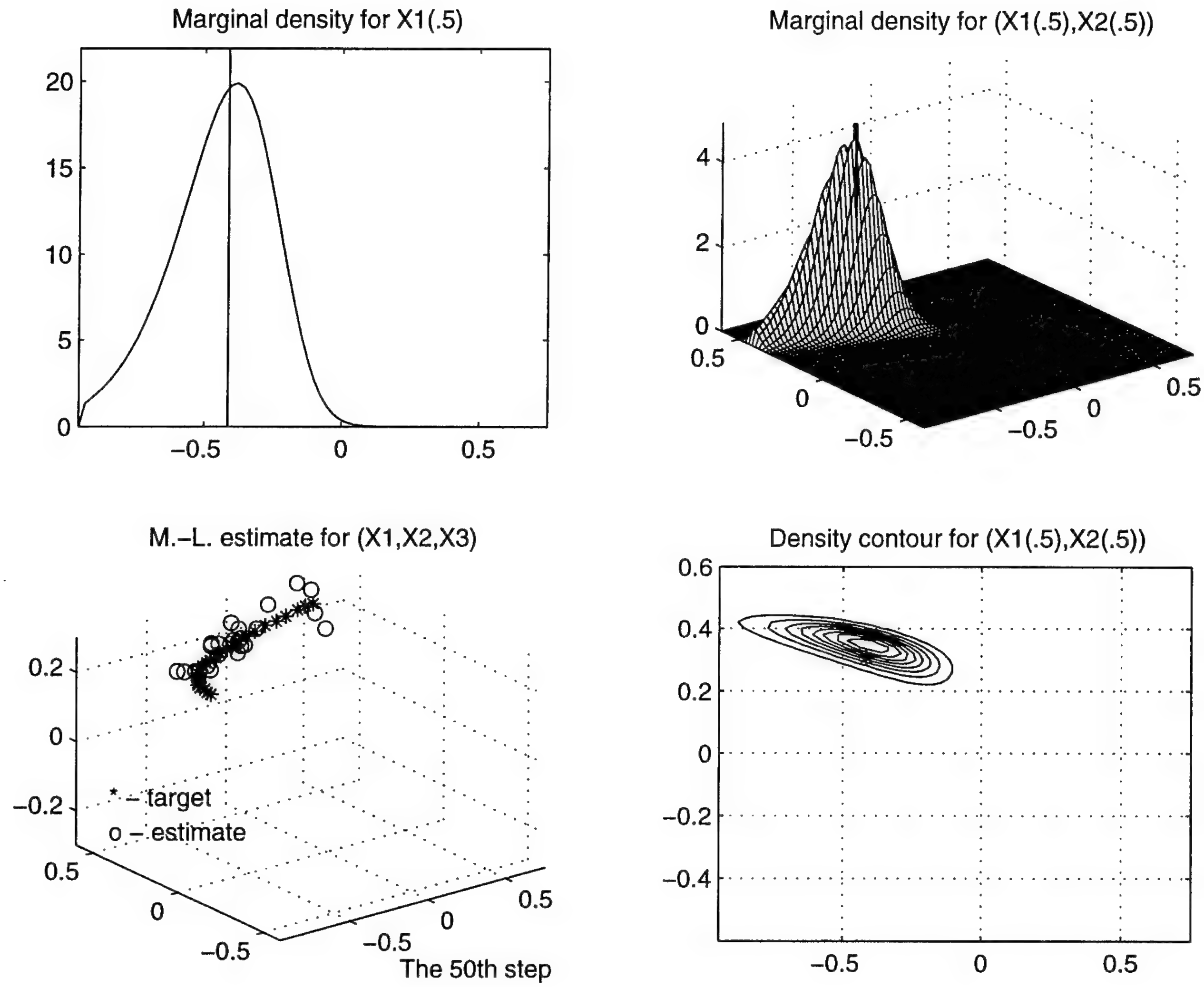
Observations without any noise in the system

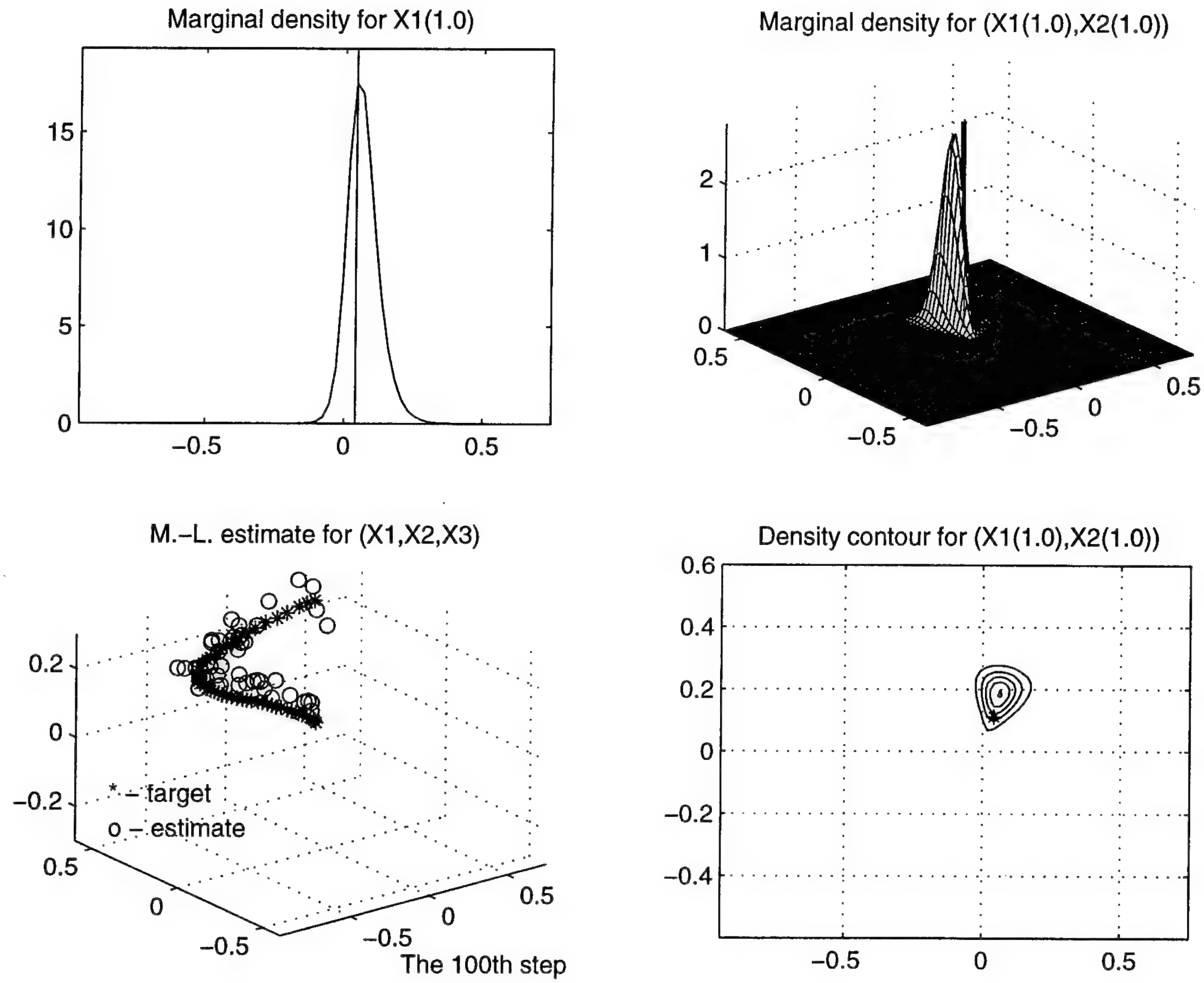



Angle–Only Observations
Fisrt Angle
TIME t -->
Second Angle
TIME t -->

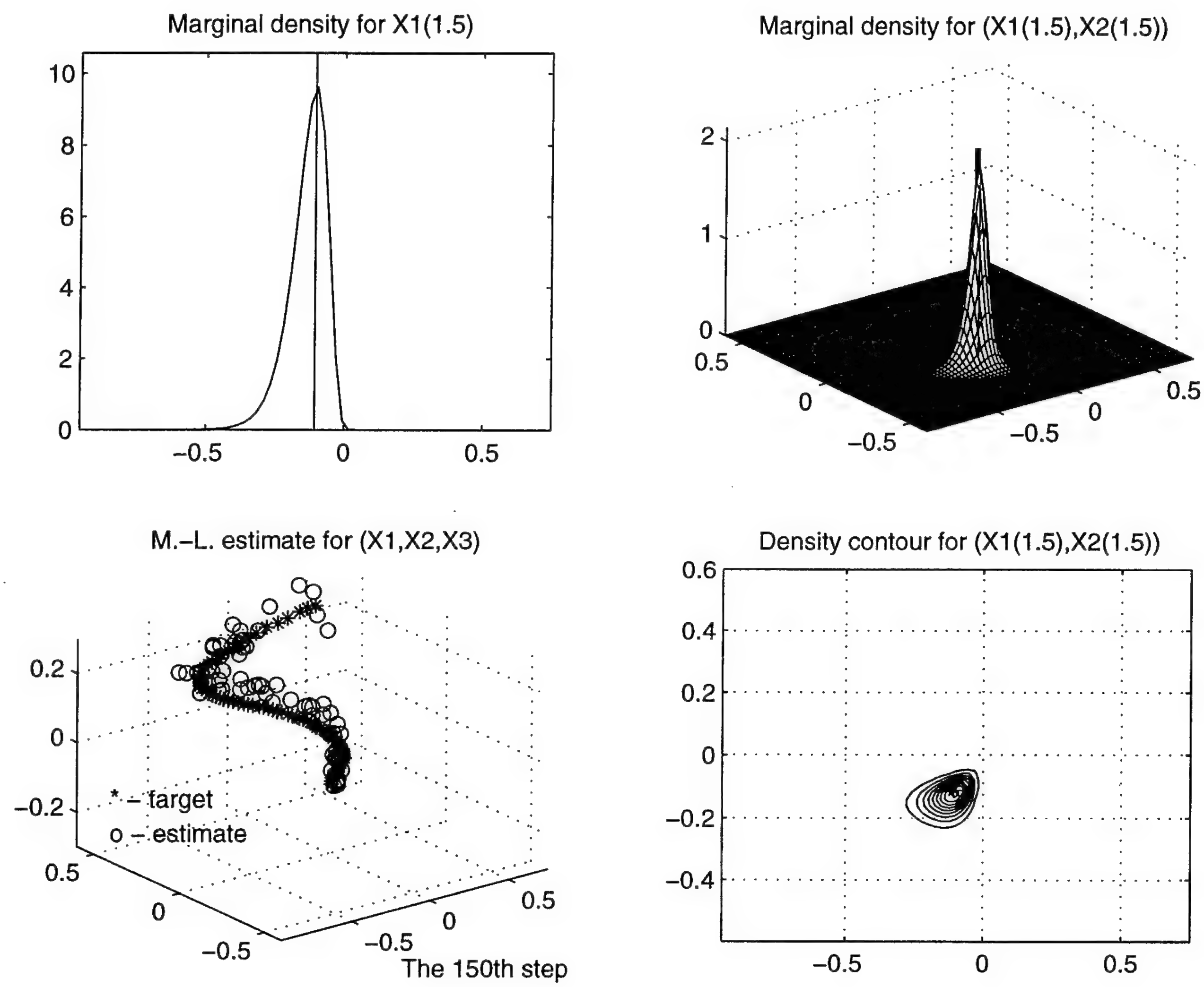


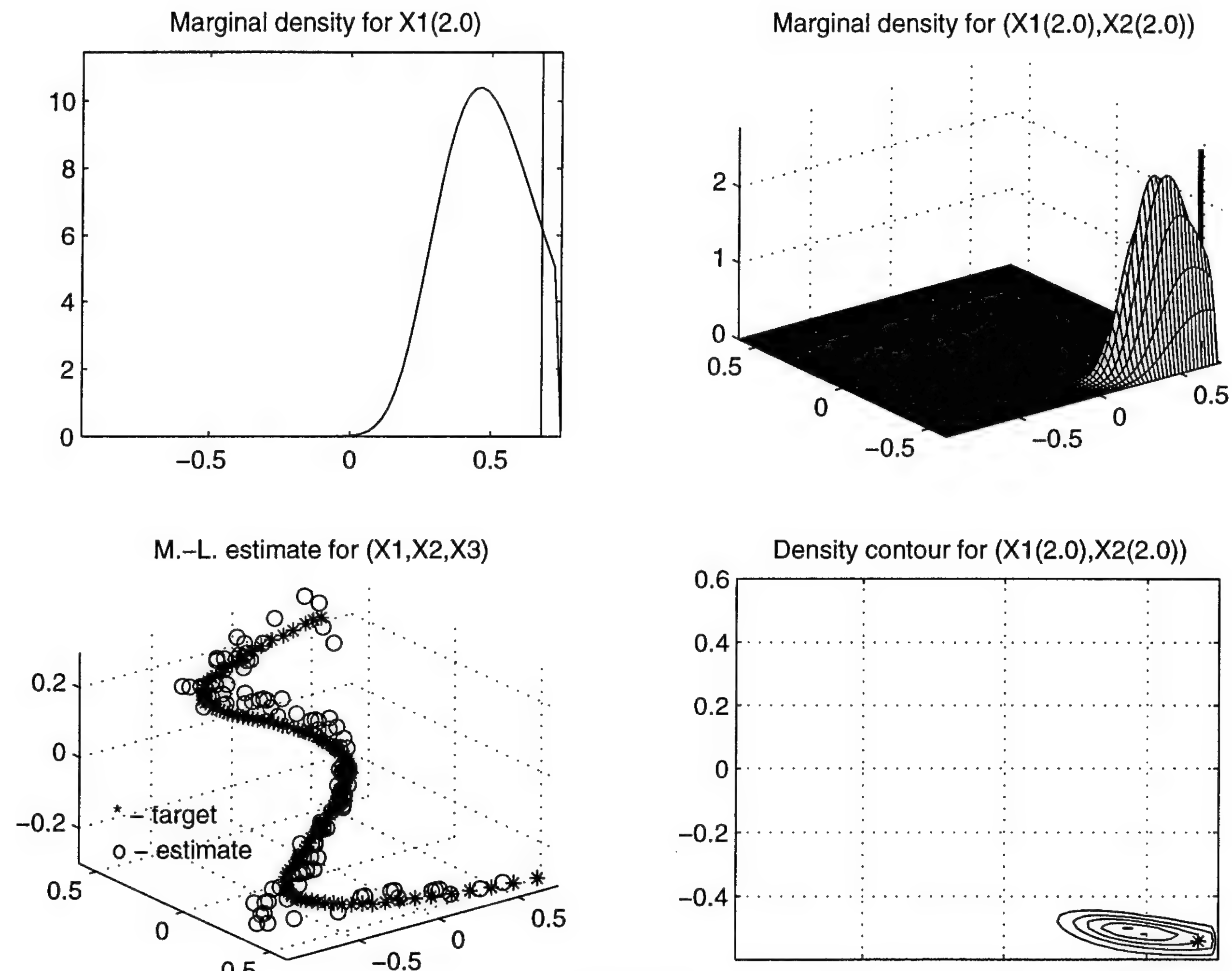
Marginal density for X1(0)
150
100
50
0
−0.5
0
0.5
Marginal density for (X1(0),X2(0))
10
5
0
0.5
0
−0.5
−0.5
0
0.5
Initial value of (X1,X2,X3)
0.2
0
−0.2
*: target
o: estimate
0.5
0
−0.5
−0.5
0
0.5
Density contour for (X1(0),X2(0))
0.6
0.4
0.2
0
−0.2
−0.4
−0.5
0
0.5


Marginal density for X1(.5)
20
15
10
5
0
-0.5
0
0.5
Marginal density for (X1(.5),X2(.5))
4
2
0
0.5
0
-0.5
-0.5
0
0.5
M.-L. estimate for (X1,X2,X3)
0.2
0
-0.2
* - target
o - estimate
0.5
0
-0.5
-0.5
0
0.5
The 50th step
Density contour for (X1(.5),X2(.5))
0.6
0.4
0.2
0
-0.2
-0.4
-0.5
0
0.5



Marginal density for X1(1.0)
15
10
5
0
−0.5
0
0.5
Marginal density for (X1(1.0),X2(1.0))
2
1
0
0.5
0
−0.5
−0.5
0
0.5
M.–L. estimate for (X1,X2,X3)
0.2
0
−0.2
* – target
o – estimate
0.5
0
−0.5
−0.5
0
0.5
The 100th step
Density contour for (X1(1.0),X2(1.0))
0.6
0.4
0.2
0
−0.2
−0.4
−0.5
0
0.5


Marginal density for X1(1.5)
10
8
6
4
2
0
−0.5
0
0.5
Marginal density for (X1(1.5),X2(1.5))
2
1
0
0.5
0
−0.5
0.5
0
−0.5
M.−L. estimate for (X1,X2,X3)
0.2
0
−0.2
* − target
o − estimate
0.5
0
−0.5
0.5
0
−0.5
The 150th step
Density contour for (X1(1.5),X2(1.5))
0.6
0.4
0.2
0
−0.2
−0.4
−0.5
0
0.5



The last step

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%       New operator splitting method for solving
%%       dX = b(X)dt + sigmaa(X)dW,  X(0) = m0
%%       z(k) = h(X(t_k)) + delta(k)v(k)
%%       Chuanxia Rao
%%       January 1996
%%       January 1997
%%       May 1997
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%       This is the main program.
%%------------------------------------
%%       Dependencies:
%%       initialize.m
%%       moreinit.m
%%       generate.m
%%       online.m
%%       plotting.m
%%------------------------------------
%%       Output:
%%       graphs
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

clear;

        count = flops;
        cpu = cputime;
initialize;                 %% initial data
moreinit;                   %% further initialization
generate;                   %% to generate processes X and Z
        flops_off = flops - count
        cpu_off = cputime - cpu

figure;
        xyz = X(1,:);
        plotting(xyz, pk);
kstart=1;       kstop=Kmax;
%kstart=Kmax+1; kstop=Kfinal-1;

        count = flops;
        cpu = cputime;
%%global time;
for ktime = kstart:kstop,

        k1 = ktime + 1;
%%      time = t0 + dt*ktime;
        fprintf(1, '%c', '  step ');
        fprintf(1, '%d\n', ktime);

        online;

kplot = ktime/plot_step;
if ktime==1 | kplot == fix(kplot),
        xyz = X(k1,:);
        plotting(xyz, pk);
end

end
        count = flops - count;
        cpu = cputime - cpu;
        cpu_on1 = cpu / (kstop-kstart+1)
        flops_on1 = count / (kstop-kstart+1)
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%      Initial data for the cont.-discr. model:
%%      dX = b(X)dt + sigmaa*dW
%%      X(0) ~ N(m0,var0)
%%      z(k) = h(X(t_k)) + delta*v(k)
%%----------------------------
%%      Chuanxia Rao
%%      Februay 1996
%%      Februry 1997
%%      June 1997
%%----------------------------
%%      Called in:
%%      all other parts
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

global  dim dim_obs
global  nx ny nz
global  xa ya za
global  xb yb zb
global  m0 var0_inv

global substeps
global small %large
global resol_vert
global method

global  denom_obs
global  denom
global  xx yy zz
%global  nxp1 nyp1 nzp1


        disp(' Initialization ...');
        dim = 3;
        dim_obs = 2;

%%  Part 1.
Tfinal = 2.0;
Kfinal = 256; Kt = Kfinal;
Kmax = 128; %64; %256;
%Kmax = Kfinal - 1;
%Kmax = Kfinal / 4;

nx = 72; %60; %40; %80;
ny = 60; %45; %30; %60;
nz = 30; %24; %20; %30;
xa = -0.95;  xb = 0.75;
ya = -0.60;  yb = 0.60;
za = -0.30;  zb = 0.30;
%xa = [-0.95, -0.6, -0.3];
%xb = [0.75, 0.6, 0.3];
%Nx = [nx,ny,nz];
%Lx = [3,2,1];
Lx = [1,1,1];                   %% bandwidth of local propogation:
                                %% It depends on (dx*dx)/(dt*sigmaa^2).

plot_step = 2; %1;              %% plot in every plot_step steps
resol_vert = 0.01;              %% for plotting target position

substeps = 1;
small = 1e-16;                  %% threshold for positive probability
%large = -log(small);
```

```
%%  Part 2.
%sigmaa = [0.0, 0.0, 0.0];          %% coefficients in signal noise
sigmaa = [0.045, 0.023, 0.012];
%delta = [0.0, 0.0];                %% coefficients in observ. noise
%delta = [0.24, 0.12];
delta = [0.64, 0.36];

m0 = [0.45, 0.42, 0.20];            %% initial mean
var0_inv = [100, 121, 64];          %% initial variance, its inverse
%var0_inv = [82, 100, 64];

%X0 = [0.75, -0.5, -0.25];
X0 = [3.6/4/2, 1./2, 1./4];
Z0 = [0,0];


%%  Part 3.
method = 11;
        %01: cdsplit                %% convection-diffusion splitting (cd)
        %02: dcsplit                %% convection-diffusion splitting (dc)
        %11: cdcsplit               %% convection-diffusion splitting (cdc)
        %12: dcdsplit               %% convection-diffusion splitting (dcd)
        %21: central1               %% central difference + backward Euler
        %22: central2               %% central difference + C-N (trapezoid)
        %31: upwind                 %% 1st order upwind + backward Euler
        %32: new2nd                 %% new "upwind" (2nd order in space & time,
                                    %%      and resulting in d.d. tridiagonals)


%%
%%  Further initialization:
%%  -- no changes needed --
%%

%%  Part 4.
dt = Tfinal/Kfinal;
dx = (xb-xa)/nx;
dy = (yb-ya)/ny;
dz = (zb-za)/nz;

dt2 = dt/2;
dt4 = dt/4;
%nxp1 = nx + 1;
%nyp1 = ny + 1;
%nzp1 = nz + 1;

denom_obs = 2 * delta.^2;
denom = (2*dt) * sigmaa.^2;
%const_norm = sigmaa... * sqrt(2*pi*dt)^dim;
diffus = dt2 * sigmaa.^2 / 2;
diffux = diffus(1) /dx/dx;
diffuy = diffus(2) /dy/dy;
diffuz = diffus(3) /dz/dz;

        xx = xa + dx * (0:nx);
        yy = ya + dy * (0:ny);
        zz = za + dx * (0:nz);
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%      Generate a 3D trajectory for
%%      dX = b(X)dt + sigmaa*dW
%%      X(0) = X0
%%      And generate an observation for
%%      z(k) = h(X(t_k)) + delta*v(k)
%%      Chuanxia Rao
%%      January 1996
%%      January 1997
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%      Dependencies:
%%      initialize.m;
%%      func_b.m (function)
%%      func_h.m (function)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%      Output:
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%       initialize;
        disp(' Generating the trajectory ...');

%out_fileZ = 'data_Z.m';
%out_fileX = 'data_X.m';

%Kfinal = Kt;
srdt = sqrt(dt);
X(1,:) = X0;
Z(1,:) = Z0;

        randn('seed', 0);
        v = randn(Kfinal,2);

        Xk = X(1,:);
for k = 1:Kfinal,
        k1 = k + 1;

        [bk1,bk2,bk3] = func_b(Xk(1),Xk(2),Xk(3));
        bk = [bk1,bk2,bk3];
        temp = Xk + dt*bk;                  %% forward Euler
        [bk1,bk2,bk3] = func_b(temp(1), temp(2), temp(3));
        b2 = bk + [bk1,bk2,bk3];
        temp = Xk + dt2 * b2;               %% trapezoid
        temp1 = srdt*diag(sigmaa)*randn(dim,1);
        X(k1,:) = temp + temp1';

        Xk = X(k1,:);
        [h1,h2] = func_h(Xk(1),Xk(2),Xk(3));
        temp2 = diag(delta)*v(k,:)';
        Z(k1,:) = [h1,h2] + temp2';
end

%%
%%      Output -- graph:
```

```
%%

        k1 = 1:(Kfinal+1);
        k11 = (1:Kfinal)+1;
        t1 = 0:dt:Tfinal;
        t11 = dt:dt:Tfinal;
        a1 = min(X(k1,1)); b1 = max(X(k1,1));
        a2 = min(X(k1,2)); b2 = max(X(k1,2));
        a3 = min(X(k1,3)); b3 = max(X(k1,3));
        aZ1 = min(Z(k11,1)); bZ1 = max(Z(k11,1));
        aZ2 = min(Z(k11,2)); bZ2 = max(Z(k11,2));

        figure;
        subplot(221), plot(t1,X(k1,1));
        axis([0,Tfinal, a1,b1]);
        subplot(222), plot(t1,X(k1,2));
        axis([0,Tfinal, a2,b2]);
        subplot(223), plot(t1,X(k1,3));
        axis([0,Tfinal, a3,b3]);
        subplot(224), plot3(X(k1,1), X(k1,2), X(k1,3), 'r');
        axis([a1,b1, a2,b2, a3,b3]);

        figure;
        subplot(211), plot(t11,Z(k11,1));
        axis([0,Tfinal, aZ1,bZ1]);
        subplot(212), plot(t11,Z(k11,2));
        axis([0,Tfinal, aZ2,bZ2]);

clear temp temp1 temp2
clear bk bk1 bk2 bk3
clear h1 h2
clear v srdt
clear a1 a2 a3 aZ1 aZ2
clear b1 b2 b3 bZ1 bZ2
clear k1 k11
clear t1 t11
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%      Further initialization for the new ADI for
%%      dX = b(X)dt + sigmaa(X)dW,   X(0) ~ N(m0,var0)
%%      Z(k) = h(X(t_k)) + delta(k)v(k)
%%----------------------------------------------------
%%      Chuanxia Rao
%%      May 1997
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%      Dependencies:
%%      initialize.m
%%------------------------------
%%      Used in:
%%      online.m & run.m
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

global  denom_obs
%global  denom
%global  nxp1 nyp1 nzp1
global  xx yy zz

global  coef0ax coef0bx coef0cx
global  coef0ay coef0by coef0cy
global  coef0az coef0bz coef0cz
global  coef1ax coef1bx coef1cx
global  coef1ay coef1by coef1cy
global  coef1az coef1bz coef1cz

%initialize;
        disp(' Computation of the coefficients used in each step')
        disp(' and setting up of the initial density ...')

        [hh1, hh2] = func_h(xx, yy, zz);
        pk = func_p0(xx, yy, zz);
%       pk = cutsmall(pk);
%       hh1 = cutsmall(hh1);
%       hh2 = cutsmall(hh2);
%%      hh = [hh1, hh2];        %% ???

%%
%%  Convection & diffusion coefficients:
%%  OR one-dimensional coefficients:
%%

        xx1 = xx(2:nx); %% xa + dx*(1:nx-1);
        yy1 = yy(2:ny); %% ya + dy*(1:ny-1);
        zz1 = zz(2:nz); %% za + dx*(1:nz-1);

if (method==01 | method==02 | method==11 | method==12),

%%      C-D splitting stuff goes here ...
        disp(' Warning: program not finished for C-D splitting ?!');

elseif (method==21 | method==22 | method==31 | method==32),

[temp_bx,temp_by,temp_bz] = func_b(xx1, yy1, zz1);
[temp_dbx,temp_dby,temp_dbz] = func_db(xx1, yy1, zz1);
temp_dbx = temp_dbx * dt2;
temp_dby = temp_dby * dt2;
temp_dbz = temp_dbz * dt2;

if (method==22),                        %% central2
        temp_bx = temp_bx * dt4/dx;
```

```
        temp_by = temp_by * dt4/dy;
        temp_bz = temp_bz * dt4/dz;
        temp_bxp = zeros(size(temp_bx));
        temp_bxn = temp_bxp;
        temp_byp = zeros(size(temp_by));
        temp_byn = temp_byp;
        temp_bzp = zeros(size(temp_bz));
        temp_bzn = temp_bzp;
elseif (method==32),                       %% new2nd
        temp_bx = temp_bx * dt2/dx;
        temp_by = temp_by * dt2/dy;
        temp_bz = temp_bz * dt2/dz;
        temp_bxp = temp_bx .* (temp_bx>0);
        temp_bxn = temp_bx .* (temp_bx<0);
        temp_byp = temp_by .* (temp_by>0);
        temp_byn = temp_by .* (temp_by<0);
        temp_bzp = temp_bz .* (temp_bz>0);
        temp_bzn = temp_bz .* (temp_bz<0);
        temp_bx = abs(temp_bx);
        temp_by = abs(temp_by);
        temp_bz = abs(temp_bz);
elseif (method==31),                       %% upwind
        temp_bx = temp_bx * dt/dx;
        temp_by = temp_by * dt/dy;
        temp_bz = temp_bz * dt/dz;
        temp_bxp = temp_bx .* (temp_bx>0);
        temp_bxn = temp_bx .* (temp_bx<0);
        temp_byp = temp_by .* (temp_by>0);
        temp_byn = temp_by .* (temp_by<0);
        temp_bzp = temp_bz .* (temp_bz>0);
        temp_bzn = temp_bz .* (temp_bz<0);
        temp_bx = abs(temp_bx);
        temp_by = abs(temp_by);
        temp_bz = abs(temp_bz);
        temp_dbx = temp_dbx * 2;
        temp_dby = temp_dby * 2;
        temp_dbz = temp_dbz * 2;
        diffux = diffux * 2;
        diffuy = diffuy * 2;
        diffuz = diffuz * 2;
elseif (method==21),               %% central1
        disp(' Warning: program not finished for central1 ?!');
end

[coef0ax,coef0bx,coef0cx, coef1ax,coef1bx,coef1cx] =...
        discret(method, diffux, temp_bx,temp_dbx, temp_bxp,temp_bxn);
[coef0ay,coef0by,coef0cy, coef1ay,coef1by,coef1cy] =...
        discret(method, diffuy, temp_by,temp_dby, temp_byp,temp_byn);
[coef0az,coef0bz,coef0cz, coef1az,coef1bz,coef1cz] =...
        discret(method, diffuz, temp_bz,temp_dbz, temp_bzp,temp_bzn);

else                               %% others
        disp(' Discretization method not known ?!')

end

clear  diffus diffux diffuy diffuz
clear  temp_bx temp_by temp_bz
clear  temp_bxp temp_byp temp_bzp
clear  temp_bxn temp_byn temp_bzn
clear  temp_dbx temp_dby temp_dbz
clear  xx1 yy1 zz1
clear  dt4 dx dy dz
```

```
function  [a0,b0,c0, a1,b1,c1] = discret(method, dif, b, db, bp, bn)
%usage:   [a0,b0,c0, a1,b1,c1] = discret(method, dif, b, db, bp, bn)

if(method==22),                 %% central2
        temp = 2*dif + db;
        a0 = dif + b;
        b0 = 1 - temp;
        c0 = dif - b;
        a1 = -a0;
        b1 = 1 + temp;
        c1 = -c0;

elseif(method==32),             %% new2nd
        temp1 = 1 + b;
        temp2 = 2*dif + db;
        difn = - dif;
        a0 = dif + bn;
        b0 = temp1 - temp2;
        c0 = dif - bp;
        a1 = difn - bp;
        b1 = temp1 + temp2;
        c1 = difn + bn;

elseif(method==31),             %% upwind
        temp1 = 1 + b;
        temp2 = 2*dif + db;
        difn = - dif;
        a1 = difn - bp;
        b1 = temp1 + temp2;
        c1 = difn + bn;
        a0 = a1;
        b0 = b1;
        c0 = c1;

elseif(method==21),             %% central1
        disp(' Warning: program not finished for central1 ?!');

else                            %% others
        disp(' Discretization method not known ?!')

end
return
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%      New ADI method for solving
%%      dX = b(X)dt + sigmaa(X)dW,  X(0) ~ N(m0,var0)
%%      Z(k) = h(X(t_k)) + delta(k)v(k)
%%---------------------------------------------------
%%      Chuanxia Rao
%%      May 1997
%%      Mar 1997
%%      Feb 1996
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%      Dependencies:
%%      initialize.m
%%      moreinit.m
%%      obs_cor.m
%%      onestep.m
%%      cutsmall.m
%%      generate.m (for observations Z)
%%---------------------------------------------------
%%      Output:
%%      posterior density pk1 (denoted as pk)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%initialize;
%%moreinit;
        %% -- They must be called before running this file.

        z1 = Z(k1,1);
        z2 = Z(k1,2);
        alpha_k1 = obs_cor(z1,z2, hh1,hh2);
%       alpha_k1 = cutsmall(alpha_k1);
                                                    %% -- corrector alpha_k1
        Tpk = onestep(pk);
%       Tpk = cutsmall(Tpk);
                                                    %% -- prior density Tpk
        pk = alpha_k1 .* Tpk;
        pk_max = max( max( max(pk,[],3),[],2 ),[],1 );
        if pk_max > 0,
        pk = pk/pk_max;
        end
        pk = cutsmall(pk);
                                                    %% -- density at step k1
clear  alpha_k1 Tpk
```

```
function  result = onestep(method, u)
%usage:   result = onestep(method, u)
%%       Chuanxia Rao
%%       Jun 1997
%%       Mar 1997
%%       Feb 1996
%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%       Dependencies:
%%       moreinit.m
%%       sol_expl.m
%%       sol_impl.m
%%%%%%%%%%%%%%%%%%%%%%%%%%%%

if(method == 01),
%        result = split_cd(
         disp('Warning: Program for splitting not finished ?!');
elseif(method == 02),
%        result = split_dc(
         disp('Warning: Program for splitting not finished ?!');
elseif(method == 11),
%        result = split_cdc(
         disp('Warning: Program for splitting not finished ?!');
elseif(method == 12),
%        result = split_dcd(
         disp('Warning: Program for splitting not finished ?!');
else

if(method == 22 | method == 32),        %% second order in time
global  coef0ax coef0bx coef0cx
global  coef0ay coef0by coef0cy
global  coef0az coef0bz coef0cz

%% 1.1  (I+A0x) u:
        u = sol_expl(1, coef0ax,coef0bx,coef0cx, u);

%% 1.2  (I+A0y) u:
        u = sol_expl(2, coef0ay,coef0by,coef0cy, u);

%% 1.3  (I+A0z) u:
        u = sol_expl(3, coef0az,coef0bz,coef0cz, u);
end

global  coef1ax coef1bx coef1cx
global  coef1ay coef1by coef1cy
global  coef1az coef1bz coef1cz

%% 2.3  (I-A1z)^{-1} u:
        u = sol_impl(3, coef1az,coef1bz,coef1cz, u);

%% 2.2  (I-A1y)^{-1} u:
        u = sol_impl(2, coef1ay,coef1by,coef1cy, u);

%% 2.1  (I-A1x)^{-1} u:
        u = sol_impl(1, coef1ax,coef1bx,coef1cx, u);

        result = u;
end
return
```

```
function  result = sol_expl(i_dir, a, b, c, u)
% usage:  result = sol_expl(i_dir, a, b, c, u)
% where   i_dir is the direction to go (1<=i_dir<=3);
%         a, b, c are three-dimensional arrays:
%         b is the diagonal and
%         u is the right-hand side.

%%      Chuanxia Rao
%%      May 1997
%%      Feb 1996

global  nx ny nz
%global  xx yy zz
%global  time
%global  nxp1 nyp1 nzp1

        [nxm1, nym1, nzm1] = size(b);
if (nxm1==nx-1 & nym1==ny-1 & nzm1==nz-1),
        result = u;
%       utemp = zeros(b);
        utemp = zeros(nxm1,nym1,nzm1);
if (i_dir==1),
        for i=1:nxm1,
        utemp(i,:,:) = a(i,:,:) .* u(i,  2:ny,2:nz)...
                     + b(i,:,:) .* u(i+1,2:ny,2:nz)...
                     + c(i,:,:) .* u(i+2,2:ny,2:nz);
        end
elseif (i_dir==2),
        for j=1:nym1,
        utemp(:,j,:) = a(:,j,:) .* u(2:nx,j,  2:nz)...
                     + b(:,j,:) .* u(2:nx,j+1,2:nz)...
                     + c(:,j,:) .* u(2:nx,j+2,2:nz);
        end
elseif (i_dir==3),
        for k=1:nzm1,
        utemp(:,:,k) = a(:,:,k) .* u(2:nx,2:ny,k  )...
                     + b(:,:,k) .* u(2:nx,2:ny,k+1)...
                     + c(:,:,k) .* u(2:nx,2:ny,k+2);
        end
else
        disp(' i_dir is beyond 1,2,3.')
end
        result(2:nx,2:ny,2:nz) = utemp;

%if (i_dir==1),                  %% for nonhomogeneous boubdary conditions
%elseif (i_dir==2),
%elseif (i_dir==3),
%       result(2:nx,2:ny, 1)  = func_u(time, xx(2:nx),yy(2:ny),zz(1));
%       result(2:nx,2:ny,nzp1)= func_u(time, xx(2:nx),yy(2:ny),zz(nzp1));
%end

else
        disp(' Dimension problem in sol_expl.m')
end
```

```
function  result = sol_impl(i_dir, a, b, c, u)
% usage:  result = sol_impl(i_dir, a, b, c, u)
% where   i_dir is the direction to go (1<=i_dir<=3);
%         a, b, c are three-dimensional arrays:
%         b is the diagonal and
%         u is the right-hand side.

%%      Chuanxia Rao
%%      May 1997
%%      Feb 1996

global  nx ny nz
%global  nxp1 nyp1 nzp1

        result = u;
        utemp = u(2:nx,2:ny,2:nz);
%if (i_dir==1),          %% for nonhomogeneous boubdary conditions
%elseif (i_dir==2),
%elseif (i_dir==3),
%       utemp(:,:,1)  = utemp(:,:,1) - a(:,:, 1) .* u(2:nx,2:ny,1);
%       utemp(:,:,nzm1) = utemp(:,:,nzm1)- c(:,:,nzm1) .* u(2:nx,2:ny,nzp1)
;
%end
        utemp = tridiag3d(i_dir, a,b,c, utemp);
        result(2:nx,2:ny,2:nz) = utemp;
```

```
function  result = tridiag3d(i_dir, a, b, c, d)
% usage:  result = tridiag3d(i_dir, a, b, c, d)
% where   i_dir is the direction to go (1<=i_dir<=3);
%         a, b, c, d are three-dimensional arrays of the same size,
%         b is the diagonal, and d is the right-hand side.
%         (d is changed after the call) ? --- No.

%%      Chuanxia Rao
%%      May 1997
%%      Feb 1996

%global  dim
%dim = 3;

na = size(a);
nb = size(b);
nc = size(c);
nd = size(d);

if(na==nb & nb==nc & nc==nd & length(nd)<=3)
        n = na(i_dir);
if(i_dir==1)
        v(1,:,:) = b(1,:,:);
        for i=2:n,
        xmult = a(i,:,:) ./ v(i-1,:,:);
        v(i,:,:) = b(i,:,:) - xmult .* c(i-1,:,:);
        d(i,:,:) = d(i,:,:) - xmult .* d(i-1,:,:);
        end
        result(n,:,:) = d(n,:,:) ./ v(n,:,:);
        for i=(n-1):-1:1,
        d(i,:,:) = d(i,:,:) - c(i,:,:) .* result(i+1,:,:);
        result(i,:,:) = d(i,:,:) ./ v(i,:,:);
        end
elseif(i_dir==2)
        v(:,1,:) = b(:,1,:);
        for i=2:n,
        xmult = a(:,i,:) ./ v(:,i-1,:);
        v(:,i,:) = b(:,i,:) - xmult .* c(:,i-1,:);
        d(:,i,:) = d(:,i,:) - xmult .* d(:,i-1,:);
        end
        result(:,n,:) = d(:,n,:) ./ v(:,n,:);
        for i=(n-1):-1:1,
        d(:,i,:) = d(:,i,:) - c(:,i,:) .* result(:,i+1,:);
        result(:,i,:) = d(:,i,:) ./ v(:,i,:);
        end
elseif(i_dir==3)
        v(:,:,1) = b(:,:,1);
        for i=2:n,
        xmult = a(:,:,i) ./ v(:,:,i-1);
        v(:,:,i) = b(:,:,i) - xmult .* c(:,:,i-1);
        d(:,:,i) = d(:,:,i) - xmult .* d(:,:,i-1);
        end
        result(:,:,n) = d(:,:,n) ./ v(:,:,n);
```

```
        for i=(n-1):-1:1,
        d(:,:,i) = d(:,:,i) - c(:,:,i) .* result(:,:,i+1);
        result(:,:,i) = d(:,:,i) ./ v(:,:,i);
        end
else
        disp(' i_dir is beyond 1,2,3.')
end
elseif( length(nd)>3 )
        disp(' dimension > 3')
else
        disp(' Dimensions do not agree in tridiag3d.')
end
```

```
function  result = obs_cor(z1,z2, hh1,hh2)
% usage:  result = obs_cor(z1,z2, hh1,hh2)
% usage:  result = obs_cor(z, hh)
% where   z is the measurement, and
%         hh is the part without noise.
%         dim = 3; dim_obs = 2;

%%      Chuanxia Rao
%%      May 1997
%%      Feb 1996
%%      Used in: online.m

global denom_obs         %% defined in moreinit.m

        temp = z1 - hh1(:,:,:);
        temp1 = temp/denom_obs(1) .* temp;
        temp = z2 - hh2(:,:,:);
        temp1 = temp1 + temp/denom_obs(2) .* temp;
        temp1 = temp1 - min( min( min(temp1,[],3),[],2 ),[],1 );
        result = exp( - temp1 );
```

```
function  plotting(xyz, pk)
%usage:   plotting(xyz, pk)
%         where length(xyz) = dim(=3)
%               size(pk) = [nxp1 nyp1 nzp1]

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%
%%      This is for plotting the tracking process.
%%      Chuanxia Rao
%%      April 1997
%%      May 1997
%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

global  xa ya za
global  xb yb zb
global  xx yy zz
global  resol_vert

%figure

        x = xyz(1);
        y = xyz(2);
        z = xyz(3);
        pk_xy = sum(pk, 3);
        pk_x = sum(pk_xy, 2);

subplot(221)
        plot(xx, pk_x);
        hold on;

        lower = 0;
        upper = max(pk_x);
        upper = upper * 1.1;
        resol = resol_vert * upper;
        vert = lower:resol:upper;
        plot(x*ones(size(vert)), vert, 'g');
        axis([xa xb lower upper]);
        pause(1);
        hold off;

subplot(222)
        pk_xy = pk_xy';
        mesh(xx, yy, pk_xy);
        hold on;

        m = max(max(pk_xy));
        m = m * 1.1;
        plot3([x x], [y y], [0 m], 'g', 'LineWidth',[2]);
        axis([xa xb ya yb 0 m]);
        pause(1);
        hold off;
```

```
subplot(224)
        contour(xx, yy, pk_xy);
        hold on;

        plot(x, y, 'g*');
        pause(1);
        hold off;

subplot(223)
        plot3(x, y, z, 'g*');
        axis([xa xb ya yb za zb]);
        hold on;

        [one, ijk] = max(pk,[],3);
        [one, ij] = max(one,[],2);
        [one, i] = max(one,[],1);
        x = xx(i);
        y = yy(ij(i));
        z = zz(ijk(i,ij(i)));
%       plot3(x, y, z, 'ro', 'LineWidth',[2]);
        plot3(x, y, z, 'r*');
        grid on;
        pause(1);
%       hold off;

return
```

```
function  result = cutsmall(array)
% usage:  result = cutsmall(array)

%%      Used in: online.m
%%      Chuanxia Rao
%%      May 1997

global small            %% defined in initialize.m

        result = (array > small) .* array;
%       result = sparse( result );
```

```
function  [x_down,y_down,z_down] = advection(dt, x_up,y_up,z_up)
% usage:  [x_down,y_down,z_down] = advection(dt, x_up,y_up,z_up)

%%      Chuanxia Rao
%%      Februry 1997
%%      April 1997
%%      June 1997

global  substeps

%dt6 = dt/6;
dt2 = dt/2;
x = x_up;
y = x_up;
z = x_up;

for k = 1:substeps,
        [bk1,bk2,bk3] = func_b(x,y,z);
        [b1,b2,b3] = func_b(x+dt*bk1,y+dt*bk2,z+dt*bk3);
        x = x + dt2 * (bk1 + b1);
        y = y + dt2 * (bk2 + b2);
        z = z + dt2 * (bk3 + b3);       %% trapezoid
%       bk = func_b(x);
%       temp = func_b(x + dt2*bk);
%       x = x + dt * temp;              %% modified Euler
%%      k1 = func_b(x);
%%      k2 = func_b(x + dt2*k1);
%%      k3 = func_b(x + dt2*k2);
%%      k4 = func_b(x + dt*k3);
%%      temp = k1 + 2*k2 + 2*k3 + k4;
%%      x = x + dt6 * temp;             %% Runge-Kutta-4
end
        x_down = x;
        y_down = y;
        z_down = z;
```

```
function  [b1,b2,b3] = func_b(x,y,z)
% usage:  [b1,b2,b3] = func_b(x,y,z)
% where   x,y,z are vectors.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%      The signal propogation function
%%      Chuanxia Rao
%%      Februry 1997
%%      May 1997
%%----------------------------------------
%%      Called in:
%%      moreinit.m
%%      generate.m
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

for j=1:length(y),
for k=1:length(z),
%       b1(1:length(x), j, k) = 200*y(j)^3 + 50*z(k) - 50;
        b1(1:length(x), j, k) = - 200*y(j)^3 + 50*z(k);
end
end
        b1 = b1/2;

%       b2 = 1./2 * ones(size(b1));
        b2 = -1./2 * ones(size(b1));

%       b3 = 1./4 * ones(size(b1));
        b3 = -1./4 * ones(size(b1));

%       b = [b1, b2, b3];
```

```
function  [db1,db2,db3] = func_db(x,y,z)
% usage:  [db1,db2,db3] = func_db(x,y,z)
% where   x,y,z are vectors.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%      The derivatives dxb1, dyb2, dzb3
%%      (The scalar function  Delta * b)
%%      Chuanxia Rao
%%      March 1997
%%      May 1997
%%-------------------------------------------
%%      Called in:
%%      moreinit.m
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

n1 = length(x);
n2 = length(y);
n3 = length(z);
        db1 = zeros(n1,n2,n3);
        db2 = zeros(n1,n2,n3);
        db3 = zeros(n1,n2,n3);
%n = length(x) * length(y) * length(z);
%       db1 = zeros(n,1);
%       db2 = zeros(n,1);
%       db3 = zeros(n,1);
%       db = zeros(n,1);

%%      db = db1 + db2 + db3;
%%      db1 = db / 3;
%%      db2 = db1;
%%      db3 = db1;
```

```
function  [h1,h2] = func_h(x,y,z)
% usage:  [h1,h2] = func_h(x,y,z)
% where   x,y,z are vectors.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%      The observation function
%%      Chuanxia Rao
%%      April 1997
%%      May 1997
%%----------------------------------------
%%      Called in:
%%      moreinit.m
%%      generate.m
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%      x = [x(:,1), x(:,2), x(:,3)]

for i=1:length(x),
for j=1:length(y),
        temp12 = x(i)^2 + y(j)^2;
if temp12==0,
for k=1:length(z),
        h1(i,j,k) = 0;
        h2(i,j,k) = asin( sign(z(k)) );
end
else
        arg1 = x(i) / sqrt(temp12);
        temp1 = sign(y(j)) * acos(arg1);
for k=1:length(z),
        h1(i,j,k) = temp1;
        temp = temp12 + z(k)^2;
        arg2 = z(k) / sqrt(temp);
        h2(i,j,k) = asin(arg2);
end
end
end
end

%       arg1 = x(:,1) ./ sqrt(x(:,1).^2 + x(:,2).^2);
%       arg2 = x(:,3) ./ sqrt(x(:,1).^2 + x(:,2).^2 + x(:,3).^2);
%if(x(:, 2)<0),
%%      h1 = 2*pi - acos(arg1);
%       h1 = acos(arg1);
%else
%       h1 = acos(arg1);
%end

%       h2 = temp .* h2;
%       h = [h1, h2];
```

```
function  p0 = func_p0(x,y,z)
% usage:  p0 = func_p0(x,y,z)
% where   x,y,z are vectors.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%       The initial density function
%%       Chuanxia Rao
%%       Februry 1997
%%       May 1997
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%       Called in:
%%       moreinit.m
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

global m0 var0_inv

%        temp = sqrt(2*pi)^3;
%        temp0 = temp*sqrt(var0_inv(1)*var0_inv(2)*var0_inv(3));
for i=1:length(x),
        temp1 = (x(i)-m0(1))^2 * var0_inv(1);
for j=1:length(y),
        temp2 = (y(j)-m0(2))^2 * var0_inv(2);
for k=1:length(z),
        temp3 = (z(k)-m0(3))^2 * var0_inv(3);
        temp3 = temp1 + temp2 + temp3;
        p0(i,j,k) = exp( -temp3/2 );
end
end
end
%       p0 = p0 / temp0;
        p0 = p0 / max( max( max(p0,[],3),[],2 ),[],1 );
```

```
function  b = func_b(x)
% usage:  b = func_b(x)
% where   x is a matrix of size (:, dim=3).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%      The signal propogation function
%%      Chuanxia Rao
%%      Februry 1997
%%      April 1997
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%      Called in:
%%      advection.m (function)
%%      generate.m
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%      x = [x(:,1), x(:,2), x(:,3)]

global gam

        b1 = -x(:,2);
        b2 = -exp(-gam*x(:,1)) .* x(:,2).^2 .* x(:,3);
        b3 = zeros(size(b1));

        b = [b1, b2, b3];
```

```
function  db = func_db(x)
% usage:  db = func_db(x)
% where   x is a matrix of size (:, dim=3).
%          db is a vector of size (:).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%        The scalar function  Delta * b
%%        Chuanxia Rao
%%      April 1997
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%        Called in:
%%        split_cdc.m
%%      run.m
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%        x = [x(:,1), x(:,2), x(:,3)]

global gam

%         db1 = zeros(size(x(:,1));
%         db2 = -2 * exp(-gam*x(:,1)) .* x(:,2) .* x(:,3);
%         db3 = zeros(size(x(:,3));

%         db = db1 + db2 + db3;
          db = -2 * exp(-gam*x(:,1)) .* x(:,2) .* x(:,3);
```

```
function  h = func_h(x)
% usage:  h = func_h(x)
% where   x is a matrix of size (:, dim=3).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%       The observation function
%%       Chuanxia Rao
%%       April 1997
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%       Called in:
%%       offline.m
%%       generate.m
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%       x = [x(:,1), x(:,2), x(:,3)]

global  MM H

         temp = MM + (x(:,1) - H).^2;
         h = sqrt(temp);
```

```
function  p0 = func_p0(x)
% usage:  p0 = func_p0(x)
% where   x is x is a matrix of size (:, dim=3).
%          p0 is a vector of size (:).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%       The initial density function
%%       Chuanxia Rao
%%       Februry 1997
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%       Called in:
%%       offline.m
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

global m0 var0_inv

          temp = sqrt(2*pi)^3;
          temp = temp*sqrt(var0_inv(1)*var0_inv(2)*var0_inv(3));
          temp1 = (x(:,1)-m0(1)).^2 * var0_inv(1);
          temp2 = (x(:,2)-m0(2)).^2 * var0_inv(2);
          temp3 = (x(:,3)-m0(3)).^2 * var0_inv(3);
          temp3 = temp1 + temp2 + temp3;
          p0 = exp( -temp3/2 ) / temp;
          p0 = p0 / max(p0);
```

```
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!      New ADI for convection-diffusion equations
!!      Chuanxia Rao
!!      May 1997
!!      Feb 1996
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!      Dependencies:
!!      Modules for initial data
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!      Output:
!!
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

!!
!! Modules for initial data:
!!
      MODULE init_data1
      REAL*8  xi
      INTEGER ix
      INTEGER, PARAMETER :: dimen = 1
      INTEGER, PARAMETER :: nx = 64
      INTEGER, PARAMETER :: nx1 = nx-1
!      INTEGER, PARAMETER :: nx2 = nx-2
      INTEGER, PARAMETER :: mt = 256
      END MODULE init_data1

      MODULE init_data2
      USE init_data1
      REAL*8  dt, dx
      REAL*8  dt2, dtx2
      REAL*8  t0, t1
      REAL*8  xa, xb, ya, yb, za, zb
      REAL*8  xx(0:nx)
      PARAMETER (t0=0D0, t1=1D0)
      PARAMETER (xa=-1.0D0, xb=1.0D0)
      PARAMETER (dx = (xb-xa)/nx)
!      PARAMETER (dt = 1D0/mt)
      PARAMETER (dt = (t1-t0)/mt)
      PARAMETER (dt2 = dt/2)
      PARAMETER (dtx2 = dt2/dx)
      PARAMETER ( xx = xa + dx * (/(i, i=0,nx)/) )
      END MODULE init_data2

      MODULE init_data3
      USE init_data2
      REAL*8  diffus, ddiff, dd_neg
      REAL*8  dneg, dpos
      PARAMETER ( diffus = 1D0 )
      PARAMETER ( ddiff = diffus*dtx2/dx )
      PARAMETER ( dd_neg = - ddiff )
      PARAMETER ( dneg = 1D0 - 2 * ddiff )
      PARAMETER ( dpos = 1D0 + 2 * ddiff )
      END MODULE init_data3

      MODULE coeffunc
      REAL*8,  PARAMETER :: PI = 3.14159265358979323
        CONTAINS
      FUNCTION func_b(x)
!!
!! The coefficient functions for convection (or drift) term:
!!
      REAL*8   func_b, x
!      REAL*8   x
!      REAL*8,  PARAMETER :: PI = 3.14159265358979323
```

```
      INTRINSIC  DSIN
       func_b = -3D0 * DSIN(PI * x)
!        func_b = 0D0
      RETURN
      END FUNCTION func_b

      FUNCTION func_c(x)
!!
!! The coefficient function for u (zero-th order term):
!!
!      REAL*8,  PARAMETER :: PI = 3.14159265358979323
      REAL*8    x, temp
!      REAL*8    func_c, x, temp
      INTRINSIC  DCOS
       temp = 3D0 * DCOS(PI * x)
       temp = temp - PI
       func_c = PI * temp
!        func_c = -PI * PI
!        func_c = 0
      RETURN
      END FUNCTION func_c

!      FUNCTION func_f(t,x,y,z)
!!
!! The force function f:
!!
!      REAL*8    func_f, t, x, y, z
!      RETURN
!      END FUNCTION func_f

      FUNCTION func_u(t, x)
!!
!! The true solution u(t, x):
!!
      REAL*8    func_u, t, x
!      REAL*8    t, x
!      REAL*8,  PARAMETER :: PI = 3.14159265358979323
      INTRINSIC  DSIN, DEXP
       func_u = DSIN(PI*x) * DEXP(-2*PI*PI*t)
!        func_u = DSIN(PI*x) * DEXP(-PI*PI*t)
      RETURN
      END FUNCTION func_u
      END MODULE coeffunc

      PROGRAM adi1d
!!
!! Main program begins:
!!
      USE init_data2
      USE coeffunc
      INTEGER  Kmax
      REAL*8   time
      REAL*8   u1(0:nx)
      REAL*8   u2(0:nx)
      REAL*8   t_fin, v
      REAL*8   error1, error2, value
!      REAL*8   ABS, MAX
      REAL  cputime, tarray(2), ETIME !, DTIME
      EXTERNAL ETIME !, DTIME
!      EXTERNAL func_u
      EXTERNAL march
!      EXTERNAL solution
      INTRINSIC  DABS, DMAX1, IDINT
```

```
!      func_u(t,x) = DSIN(PI*x) * DEXP(-PI*PI*t)
!      func_u(t,x) = DSIN(PI*x) * DEXP(-2*PI*PI*t)

      cputime = ETIME(tarray)
!      cputime = DTIME(tarray)


      Kmax = 2
      t_fin = Kmax*dt
!      t_fin = t1
!      CALL solution(t_fin, u1, u2)
                        !! "t_fin" might be slightly changed after the CALL.

      DO ix = 0, nx
         xi = xx(ix)
         u1(ix) = func_u(t0, xi)
         u2(ix) = func_u(t0, xi)
      END DO

!!
!! New method:
!!
      CALL march( Kmax, 1, u1 )

!!
!! Old method:
!!
      CALL march( Kmax, 2, u2 )

!      cputime = DTIME(tarray)
      cputime = ETIME(tarray) - cputime

!!
!! Maximum errors:
!!
      value = 0D0
      error1 = 0D0
      error2 = 0D0
      DO ix = 1, nx1
         xi = xx(ix)
         v = func_u(t_fin, xi)
         value = DMAX1( value, DABS(v) )
         error1 = DMAX1( error1, DABS( v - u1(ix) ) )
         error2 = DMAX1( error2, DABS( v - u2(ix) ) )
      END DO

      PRINT *, ' dx = ', dx
      PRINT *, ' dt = ', dt
      PRINT *, ' t1 = ', t1
      PRINT *, ' t_fin = ', t_fin
      PRINT *, ' total cpu = ', cputime
      PRINT *, ' max value = ', value
      PRINT *, ' max error_new = ', error1
      PRINT *, ' max error_old = ', error2

      END PROGRAM adi1d

      SUBROUTINE march(Kmax, label, u)
!      USE coefficients
      USE coeffunc
      USE init_data3
      INTEGER  label
      INTEGER  Kmax, ktime
      LOGICAL judge
```

```
      REAL*8  temp_b, temp_c, vect_b
      REAL*8  time
      REAL*8  u(0:nx)
      REAL*8  ux(nx1)
      REAL*8  coef0a(nx1)
      REAL*8  coef0b(nx1)
      REAL*8  coef0c(nx1)
      REAL*8  coef1a(nx1)
      REAL*8  coef1b(nx1)
      REAL*8  coef1c(nx1)
      REAL*8  aax(nx1), bbx(nx1), ccx(nx1), ddx(nx1)
!      REAL*8  coef1d(nx1)
!      REAL*8  f(0:nx)
!      EXTERNAL func_u
!      EXTERNAL func_b, func_c
      INTRINSIC  DABS
!      EXTERNAL coef
      INTENT(IN) :: Kmax, label
      INTENT(INOUT) :: u

!!       func_b(x) = -3D0 * DSIN(PI * x)
!      func_b(x) = 0D0
!!       func_c(x) = PI * (3D0*DCOS(PI*x) - PI)
!!       func_c(x) = -PI * PI
!      func_c(x) = 0D0

!         CALL coef(label, coef0a,coef0b,coef0c, coef1a,coef1b,coef1c)
!         CALL coef(label)
      DO ix = 1, nx1
         xi = xx(ix)
         vect_b = func_b(xi) * dtx2
         temp_c = func_c(xi) * dt2
         judge = (vect_b .LE. 0D0)
         temp_b = DABS(vect_b)

      IF (label==1) THEN
         IF (judge) THEN
            coef0a(ix) = ddiff
            coef0c(ix) = ddiff + vect_b
            coef1a(ix) = dd_neg + vect_b
            coef1c(ix) = dd_neg
         ELSE
            coef0a(ix) = ddiff - vect_b
            coef0c(ix) = ddiff
            coef1a(ix) = dd_neg
            coef1c(ix) = dd_neg - vect_b
         END IF
            coef0b(ix) = dneg +  temp_b + temp_c
            coef1b(ix) = dpos +  temp_b - temp_c
      ELSE
            vect_b = vect_b / 2
            coef0a(ix) = ddiff - vect_b
            coef0c(ix) = ddiff + vect_b
            coef1a(ix) = dd_neg + vect_b
            coef1c(ix) = dd_neg - vect_b
            coef0b(ix) = dneg + temp_c
            coef1b(ix) = dpos - temp_c
      END IF

      END DO

      DO ktime = 1, Kmax
!         time = t0 + dt*ktime
```

```
          DO ix = 1,nx1
             aax(ix) = coef1a(ix)
             bbx(ix) = coef1b(ix)
             ccx(ix) = coef1c(ix)
             ddx(ix) = coef0a(ix) * u(ix-1)  &
     &                + coef0b(ix) * u(ix)    &
     &                + coef0c(ix) * u(ix+1)
          END DO
!          u(1:nx1) = ux
!             u(0) = func_u(time, xx(0))
!             u(nx) = func_u(time, xx(nx))
!          u = u + f(ktime, :)
!                + f(ktime, ix)

!          aax = coef1a(2:nx1)
!          bbx = coef1b(1:nx1)
!          ccx = coef1c(1:nx2)
!          ddx = u(1:nx1)
!             ddx(1) = ddx(1) - coef1a(1, 1,jy,kz)*u(0,jy,kz)
!             ddx(nx1) = ddx(nx1) - coef1c(1, nx1,jy,kz)*u(nx,jy,kz)
          CALL tridiag(nx1, aax, bbx, ccx, ddx, ux)
            u(1:nx1) = ux

      END DO

      RETURN
      END SUBROUTINE march

      SUBROUTINE tridiag(N, A,B,C,D, X)
!!
!! The tridiagonal solver:
!!
      INTEGER  N
      REAL*8  A(N),B(N),C(N),D(N),X(N)
!      REAL*8  A(N-1),B(N),C(N-1), D(N), X(N)
      REAL*8  XMULT
      INTENT(IN) :: N, A,C
      INTENT(INOUT) :: B,D, X
      DO 2 I = 2,N
        XMULT = A(I)/B(I-1)
!        XMULT = A(I-1)/B(I-1)
        B(I) = B(I) - XMULT*C(I-1)
        D(I) = D(I) - XMULT*D(I-1)
   2  CONTINUE
      X(N) = D(N)/B(N)
      DO 3 I = N-1,1,-1
        X(I) = (D(I) - C(I)*X(I+1))/B(I)
   3  CONTINUE
      RETURN
      END SUBROUTINE tridiag
```

```
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!      New ADI for convection-diffusion equations
!!      Chuanxia Rao
!!      May 1997
!!      Feb 1996
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!      Dependencies:
!!      Modules for initial data
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!      Output:
!!
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

!!
!! Modules for initial data:
!!
      MODULE init_data1
      REAL*8, PARAMETER :: PI = 3.14159265358979323
      REAL*8  xi, yj, zk
      INTEGER ix, jy, kz
      INTEGER, PARAMETER :: dimen = 3
      INTEGER, PARAMETER :: nx=32, ny=32, nz=32
!      INTEGER, PARAMETER :: nx=64, ny=64, nz=64
      INTEGER, PARAMETER :: nx1=nx-1, ny1=ny-1, nz1=nz-1
      INTEGER, PARAMETER :: nx2=nx-2, ny2=ny-2, nz2=nz-2
      INTEGER, PARAMETER :: mt = 100
!      INTEGER, PARAMETER :: mt = 200
!      INTEGER, PARAMETER :: msubt = 4
      END MODULE init_data1

      MODULE init_data2
      USE init_data1
      REAL*8  diffus(dimen)
      REAL*8  dt, dx(dimen)
      REAL*8  dtdim2, dtx2(dimen)
      REAL*8  t0, t1
      REAL*8  xa, xb, ya, yb, za, zb
      REAL*8  xx(0:nx)
      REAL*8  yy(0:ny)
      REAL*8  zz(0:nz)
      PARAMETER (t0=0D0, t1=1D0)
      PARAMETER (xa=-1.0D0, xb=1.0D0)
      PARAMETER (ya=-1.0D0, yb=1.0D0)
      PARAMETER (za=-1.0D0, zb=1.0D0)
      PARAMETER ( dx = (/ (xb-xa)/nx, (yb-ya)/ny, (zb-za)/nz /) )
!      PARAMETER (dt = 1D0/mt)
      PARAMETER (dt = (t1-t0)/mt)
      PARAMETER (dtdim2 = dt/dimen/2)
      PARAMETER ( dtx2 = dt/dx/2 )
!      PARAMETER (subdt = dt/msubt)  !!(subdt = dt)
      PARAMETER ( xx = xa + dx(1) * (/(i, i=0,nx)/) )
      PARAMETER ( yy = ya + dx(2) * (/(j, j=0,ny)/) )
      PARAMETER ( zz = za + dx(3) * (/(k, k=0,nz)/) )
      PARAMETER ( diffus = (/ 1D0/6D0, 1D0/6D0, 1D0/6D0 /) )
!      PARAMETER ( diffus = (/ 1D0, 1D0, 1D0 /) )
!      DATA  diffus /1D0, 2D0, 3D0/ !! diffusion parameters
      END MODULE init_data2

      MODULE init_out
      CHARACTER (*) :: file_out !, fmt_out, status_out
      PARAMETER (file_out = 'result.temp')
      END MODULE init_out

      MODULE coeffunc
```

```
      USE init_data1
      CONTAINS
!!
!! The coefficient functions for convection (or drift) term:
!!
      FUNCTION func_b(x,y,z)
      REAL*8   func_b(dimen)
      REAL*8   x, y, z
       func_b(1) = - DSIN(PI * x)
       func_b(2) = - DSIN(PI * y)
       func_b(3) = - DSIN(PI * z)
      END FUNCTION func_b
!!
!! The coefficient function for u (zero-th order term):
!!
      FUNCTION func_c(x,y,z)
      REAL*8   func_c, x, y, z, temp
       temp = DCOS(PI * x)
       temp = temp + DCOS(PI * y)
       temp = temp + DCOS(PI * z)
       temp = temp - PI
       func_c = PI * temp
      END FUNCTION func_c
!!
!! The force function f:
!!
!      FUNCTION func_f(t,x,y,z)
!      REAL*8   func_f, t, x, y, z, temp
!      END FUNCTION func_f

      FUNCTION func_u(t, x,y,z)
      REAL*8   func_u, t, x, y, z, temp
!      REAL*8,  PARAMETER :: PI = 3.14159265358979323
      INTRINSIC  DSIN, DEXP
       temp = DSIN(PI * x)
       temp = temp * DSIN(PI * y)
       temp = temp * DSIN(PI * z)
       func_u = temp * DEXP(-3D0/2*PI*PI*t)
!       func_u = temp * DEXP(-4*PI*PI*t)
      END FUNCTION func_u
      END MODULE coeffunc

      PROGRAM adi3d
!!
!! Main program begins:
!!
      USE init_out
      USE init_data2
      USE coeffunc
      REAL*8  u(0:nx,0:ny,0:nz)
      REAL*8  t_fin, v, error, value
!      REAL*8  ABS, MAX
      REAL  cputime, tarray(2), ETIME
      EXTERNAL ETIME
      EXTERNAL solution
      INTRINSIC  DABS, DMAX1
!      INTRINSIC  ABS, EXP, SQRT, ATAN, IDINT, MAX, MATMUL,
                                  ! MAXLOC, DOT_PRODUCT

!      PRINT *, 'Off-line calculations, please wait ...'

      OPEN (UNIT=8, FILE=file_out, STATUS='REPLACE')
      WRITE (8, *) 'nx = ', nx, ';'
      WRITE (8, *) 'ny = ', ny, ';'
```

```
      WRITE (8, *) 'nz = ', nz, ';'
      WRITE (8, *) 'mt = ', mt, ';'
!      WRITE (8, '(9A)') 'index = ['
! 333  FORMAT (A4, I4, A1)

      cputime = ETIME(tarray)
!      cputime = DTIME(tarray)

!      t_fin = t1
      t_fin = 2*dt
      CALL solution(t_fin, u)
                          !! "t_fin" might be slightly changed after the CALL.

!      cputime = DTIME(tarray)
      cputime = ETIME(tarray) - cputime

      PRINT *, ' dx = ', dx
      PRINT *, ' dt = ', dt
      PRINT *, ' t1 = ', t1
      PRINT *, ' t_fin = ', t_fin
      PRINT *, ' total cpu = ', cputime
!      WRITE (8, 11) 'dx =', dx(1), ';'
      WRITE (8, *) 'dx = ', dx(1), ';'
      WRITE (8, *) 'dy = ', dx(2), ';'
      WRITE (8, *) 'dz = ', dx(3), ';'
      WRITE (8, *) 'dt = ', dt, ';'
!      WRITE (8, 22) 't1    = ', t1, ';'
      WRITE (8, *) 't1 = ', t1, ';'
      WRITE (8, *) 't_fin = ', t_fin, ';'
      WRITE (8, *) 'total_cpu = ', cputime, ';'
!      WRITE (8, 33) 'total_cpu = ', cputime, ';'
! 11   FORMAT(A4, E19.7, A1)
! 22   FORMAT(A7, F16.8, A1)
! 33   FORMAT(A11, F12.3, A1)

!!
!! Maximum error:
!!
      value = 0D0
      error = 0D0
      DO ix = 0, nx
         xi = xx(ix)
      DO jy = 0, ny
         yj = yy(jy)
      DO kz = 0, nz
         zk = zz(kz)
         v = func_u(t_fin, xi,yj,zk)
         value = DMAX1( value, DABS(v) )
         error = DMAX1( error, DABS( v - u(ix,jy,kz) ) )
      END DO
      END DO
      END DO

      PRINT *, ' max value = ', value
      PRINT *, ' max error = ', error
      WRITE (8, *) 'max_value = ', value, ';'
      WRITE (8, *) 'max_error = ', error, ';'
! 44   FORMAT(A11, F16.9, A1)
! 55   FORMAT(A11, E16.6, A1)

      CLOSE (UNIT=8)

!      CONTAINS
!!
```

```
!! The initial condition:
!!
!      FUNCTION func_u0(x,y,z)
!      REAL*8   func_u0, x, y, z, temp
!       temp = DSIN(PI * x)
!       temp = temp * DSIN(PI * y)
!       func_u0 = temp * DSIN(PI * z)
!      END FUNCTION func_u0

      END PROGRAM adi3d

      SUBROUTINE solution(t_fin, u)
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!       New ADI solver for 3D convection-diffusion equations
!!       May 9, 1997
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
      USE init_data2
      USE coeffunc
      INTEGER  ktime, Kmax
      REAL*8  t_fin, time
      REAL*8  u(0:nx, 0:ny, 0:nz)
      REAL*8  ux(nx1), uy(ny1), uz(nz1)
      REAL*8  coef0a(dimen, nx1, ny1,nz1)
      REAL*8  coef0b(dimen, nx1, ny1,nz1)
      REAL*8  coef0c(dimen, nx1, ny1,nz1)
      REAL*8  coef1a(dimen, nx1, ny1,nz1)
      REAL*8  coef1b(dimen, nx1, ny1,nz1)
      REAL*8  coef1c(dimen, nx1, ny1,nz1)
      REAL*8  aax(nx2), bbx(nx1), ccx(nx2), ddx(nx1)
      REAL*8  aay(ny2), bby(ny1), ccy(ny2), ddy(ny1)
      REAL*8  aaz(nz2), bbz(nz1), ccz(nz2), ddz(nz1)
      REAL  cpu, tarray(2), DTIME
      EXTERNAL DTIME
      EXTERNAL coef
      INTRINSIC  IDINT
      INTENT(INOUT) :: t_fin
      INTENT(OUT) :: u

      cpu = DTIME(tarray)
      CALL coef( coef0a,coef0b,coef0c, coef1a,coef1b,coef1c )

      DO ix = 0, nx
         xi = xx(ix)
      DO jy = 0, ny
         yj = yy(jy)
      DO kz = 0, nz
         zk = zz(kz)
         u(ix,jy,kz) = func_u(t0, xi,yj,zk)
      END DO
      END DO
      END DO
      cpu = DTIME(tarray)
      PRINT *, ' cpu for the coefficients and initialization = ', cpu

      Kmax = IDINT( (t_fin-t0)/dt )
      t_fin = t0 + dt*Kmax
      IF(Kmax .NE. mt) PRINT *, ' Check: t_final might be changed.'

      DO ktime = 1, Kmax
         time = t0 + dt*ktime
!!
!! 1. The explicit part, in alternating directions:
!!
         DO ix = 1,nx1
```

```
            xi = xx(ix)
         DO jy = 1,ny1
            yj = yy(jy)
         DO kz = 1,nz1
            uz(kz) = coef0a(3,ix,jy,kz) * u(ix,jy,kz-1)  &
     &                + coef0b(3,ix,jy,kz) * u(ix,jy,kz)    &
     &                + coef0c(3,ix,jy,kz) * u(ix,jy,kz+1)
         END DO
            u(ix, jy, 1:nz1) = uz
!            u(ix, jy, 0)  = func_u(time, xi, yj, zz(0))
!            u(ix, jy, nz) = func_u(time, xi, yj, zz(nz))
         END DO
         END DO

         DO kz = 1,nz1
            zk = zz(kz)
         DO ix = 1,nx1
            xi = xx(ix)
         DO jy = 1,ny1
            uy(jy) = coef0a(2,ix,jy,kz) * u(ix,jy-1,kz)  &
     &                + coef0b(2,ix,jy,kz) * u(ix,jy,kz)    &
     &                + coef0c(2,ix,jy,kz) * u(ix,jy+1,kz)
         END DO
            u(ix, 1:ny1, kz) = uy
!            u(ix, 0, kz) = func_u(time, xi, yy(0), zk)
!            u(ix, ny,kz) = func_u(time, xi, yy(ny),zk)
         END DO
         END DO

         DO jy = 1,ny1
            yj = yy(jy)
         DO kz = 1,nz1
            zk = zz(kz)
         DO ix = 1,nx1
            ux(ix) = coef0a(1,ix,jy,kz) * u(ix-1,jy,kz)  &
     &                + coef0b(1,ix,jy,kz) * u(ix,jy,kz)    &
     &                + coef0c(1,ix,jy,kz) * u(ix+1,jy,kz)
         END DO
            u(1:nx1, jy, kz) = ux
!            u(0, jy, kz) = func_u(time, xx(0), yj, zk)
!            u(nx,jy, kz) = func_u(time, xx(nx),yj, zk)
         END DO
         END DO
!         u = u + f(ktime, :,:,:)
!              + f(ktime, ix,jy,kz)

!!
!! 2. The implicit part, in alternating directions:
!!

         DO jy = 1,ny1
         DO kz = 1,nz1
            aax = coef1a(1, 2:nx1,jy,kz)
            bbx = coef1b(1, 1:nx1,jy,kz)
            ccx = coef1c(1, 1:nx2,jy,kz)
            ddx = u(1:nx1, jy, kz)
!            ddx(1) = ddx(1) - coef1a(1, 1,jy,kz)*u(0,jy,kz)
!            ddx(nx1) = ddx(nx1) - coef1c(1, nx1,jy,kz)*u(nx,jy,kz)
         CALL tridiag(nx1, aax, bbx, ccx, ddx, ux)
!       IF(ktime==1 .AND. jy==1 .AND. kz==1)            &
!     &             PRINT *, ' after calling tridiag ...'
            u(1:nx1, jy, kz) = ux
         END DO
         END DO
```

```
!      IF(ktime .EQ. 1) PRINT *, ' one implicit step survived ...'

          DO kz = 1,nz1
          DO ix = 1,nx1
             aay = coef1a(2, ix,2:ny1,kz)
             bby = coef1b(2, ix,1:ny1,kz)
             ccy = coef1c(2, ix,1:ny2,kz)
             ddy = u(ix, 1:ny1, kz)
!              ddy(1) = ddy(1) - coef1a(2, ix,1,kz)*u(ix,0,kz)
!              ddy(ny1) = ddy(ny1) - coef1c(2, ix,ny1,kz)*u(ix,ny,kz)
          CALL tridiag(ny1, aay, bby, ccy, ddy, uy)
             u(ix, 1:ny1, kz) = uy
          END DO
          END DO

          DO ix = 1,nx1
          DO jy = 1,ny1
             aaz = coef1a(3, ix,jy,2:nz1)
             bbz = coef1b(3, ix,jy,1:nz1)
             ccz = coef1c(3, ix,jy,1:nz2)
             ddz = u(ix, jy, 1:ny1)
!              ddz(1) = ddz(1) - coef1a(3, ix,jy,1)*u(ix,jy,0)
!              ddz(nz1) = ddz(nz1) - coef1c(3, ix,jy,nz1)*u(ix,jy,nz)
          CALL tridiag(nz1, aaz, bbz, ccz, ddz, uz)
             u(ix, jy, 1:nz1) = uz
          END DO
          END DO

!          WHERE (u < 0.0D0)  u = 0.0D0
!          WHERE (u(:,:,:) < 0.0D0)  u(:, :, :) = 0.0D0

      END DO

      RETURN
      END SUBROUTINE solution

      SUBROUTINE coef( coef0a,coef0b,coef0c, coef1a,coef1b,coef1c )
!      SUBROUTINE coef( coef0a,coef0b,coef0c, coef1a,coef1b,coef1c, f )
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!     The coefficients for solving the 1D sub-problems
!!     May 9, 1997
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
      USE init_data2
      USE coeffunc
      REAL*8  coef0a(dimen, nx1, ny1,nz1)
      REAL*8  coef0b(dimen, nx1, ny1,nz1)
      REAL*8  coef0c(dimen, nx1, ny1,nz1)
      REAL*8  coef1a(dimen, nx1, ny1,nz1)
      REAL*8  coef1b(dimen, nx1, ny1,nz1)
      REAL*8  coef1c(dimen, nx1, ny1,nz1)
!      REAL*8  f(0:nx, 0:ny, 0:nz)
      REAL*8  dxyz(dimen), dneg(dimen), dpos(dimen)
      REAL*8  dxyz_neg, temp, temp_c, vect_b(dimen)
      LOGICAL judge(dimen)
!      COMMON  /coef_right/  convection, diffusion, dxyz
      INTRINSIC  DABS, DSIN, DCOS
      INTENT(OUT) ::  coef0a,coef0b,coef0c, coef1a,coef1b,coef1c

      DO i = 1, dimen
         dxyz(i) = diffus(i)*dtx2(i)/dx(i)
         temp = 2 * dxyz(i)
         dneg(i) = 1 - temp
         dpos(i) = 1 + temp
```

```
        END DO

        DO ix = 1, nx1
           xi = xx(ix)
        DO jy = 1, ny1
           yj = yy(jy)
        DO kz = 1, nz1
           zk = zz(kz)

           vect_b = func_b( xi, yj, zk )
           temp_c = func_c( xi, yj, zk ) * dtdim2
           judge = (vect_b .LE. 0D0)

           DO i = 1, dimen
              temp = vect_b(i) * dtx2(i)
              temp_b = DABS(temp)
              dxyz_neg = - dxyz(i)
           IF (judge(i)) THEN
              coef0a(i, ix,jy,kz) = dxyz(i)
              coef0c(i, ix,jy,kz) = dxyz(i) + temp
              coef1a(i, ix,jy,kz) = dxyz_neg + temp
              coef1c(i, ix,jy,kz) = dxyz_neg
           ELSE
              coef0a(i, ix,jy,kz) = dxyz(i) - temp
              coef0c(i, ix,jy,kz) = dxyz(i)
              coef1a(i, ix,jy,kz) = dxyz_neg
              coef1c(i, ix,jy,kz) = dxyz_neg - temp
           END IF
              coef0b(i, ix,jy,kz) = dneg(i) +  temp_b + temp_c
              coef1b(i, ix,jy,kz) = dpos(i) +  temp_b - temp_c
           END DO

       END DO
       END DO
       END DO

       RETURN
       END SUBROUTINE coef

!       SUBROUTINE f_add(vv, jjudge, dd_judge,dd, ff)
!       REAL*8  vv, dd_judge, dd, ff
!       LOGICAL jjudge
!       INTENT(IN) :: vv, jjudge, dd_judge, dd
!       INTENT(INOUT) :: ff
!       IF(vv .NE. 0D0) THEN
!          IF(jjudge) THEN
!             ff = ff + vv * dd_judge
!          ELSE
!             ff = ff + vv * dd
!          END IF
!       END IF
!       RETURN
!       END SUBROUTINE f_add

       SUBROUTINE tridiag(N,A,B,C,D,X)
       INTEGER  N
       REAL*8  A(N-1),B(N),C(N-1),D(N),X(N)
       INTENT(IN) :: N, A,C
       INTENT(INOUT) :: B,D, X
       DO 2 I = 2,N
         XMULT = A(I-1)/B(I-1)
         B(I) = B(I) - XMULT*C(I-1)
         D(I) = D(I) - XMULT*D(I-1)
   2   CONTINUE
```

```
      X(N) = D(N)/B(N)
      DO 3 I = N-1,1,-1
        X(I) = (D(I) - C(I)*X(I+1))/B(I)
   3  CONTINUE
      RETURN
      END SUBROUTINE tridiag
```

```
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!      On-line computation of a 3-D filtering problem
!!      Chuanxia Rao
!!      January 1997
!!      April 1996
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!      Dependencies:
!!      data_h       -- computed here, simple
!!      data_v       -- offline.data
!!      data_z       -- observed.data
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

!!
!! Modules for initial data:
!!
      MODULE init_dim
      INTEGER  dimen, dim_ob
      PARAMETER (dimen = 3)
      PARAMETER (dim_ob = 2)
      END MODULE init_dim

      MODULE init_data1
      USE init_dim
      REAL*8, PARAMETER :: PI = 3.14159265358979323
      REAL*8  xi, yj, zk
      INTEGER ix, jy, kz
      INTEGER, PARAMETER :: nx=32, ny=32, nz=32
!      INTEGER, PARAMETER :: nx=64, ny=64, nz=64
      INTEGER, PARAMETER :: nx1=nx-1, ny1=ny-1, nz1=nz-1
      INTEGER, PARAMETER :: nx2=nx-2, ny2=ny-2, nz2=nz-2
      INTEGER, PARAMETER :: mt = 100
!      INTEGER, PARAMETER :: mt = 200
!      INTEGER, PARAMETER :: msubt = 4
      END MODULE init_data1

      MODULE init_data2
      USE init_data1
      REAL*8  diffus(dimen)
      REAL*8  dt, dx(dimen)
      REAL*8  dtdim2, dtx2(dimen)
      REAL*8  t0, t1
      REAL*8  xa, xb, ya, yb, za, zb
      REAL*8  xx(0:nx)
      REAL*8  yy(0:ny)
      REAL*8  zz(0:nz)
      PARAMETER (t0=0D0, t1=1D0)
      PARAMETER (xa=-1.0D0, xb=1.0D0)
      PARAMETER (ya=-1.0D0, yb=1.0D0)
      PARAMETER (za=-1.0D0, zb=1.0D0)
      PARAMETER ( dx = (/ (xb-xa)/nx, (yb-ya)/ny, (zb-za)/nz /) )
!      PARAMETER (dt = 1D0/mt)
      PARAMETER (dt = (t1-t0)/mt)
      PARAMETER (dtdim2 = dt/dimen/2)
      PARAMETER ( dtx2 = dt/dx/2 )
!      PARAMETER (subdt = dt/msubt)  !!(subdt = dt)
      PARAMETER ( xx = xa + dx(1) * (/(i, i=0,nx)/) )
      PARAMETER ( yy = ya + dx(2) * (/(j, j=0,ny)/) )
      PARAMETER ( zz = za + dx(3) * (/(k, k=0,nz)/) )
      PARAMETER ( diffus = (/ 1D0/6D0, 1D0/6D0, 1D0/6D0 /) )
!      PARAMETER ( diffus = (/ 1D0, 1D0, 1D0 /) )
!      DATA  diffus /1D0, 2D0, 3D0/ !! diffusion parameters
      END MODULE init_data2
      MODULE init_noise
      USE init_dim
```

```
      REAL*8   delta(dimen)
      REAL*8   sigma(dim_ob)
      DATA  sigma /2.3D-1, 3.0D-2/          !! noise parameters in observation
      DATA  delta /1.2D-1, 2.0D-2, 1.0D-2/ !! noise parameters in signal
      END MODULE init_noise

      MODULE init_dens
      USE init_dim
      REAL*8   xmean(dimen), var_inv(dimen,dimen)
                                 !! initial mean and variance inverse
      DATA  xmean /2.3D-1,2, 3.0D-2/
      DATA  var_inv /1.2D1,0,0, 0,2.0D2,0, 0,0,1.0D2/
      END MODULE init_dens

      MODULE init_online
      CHARACTER (*) :: file_z, fmt_z, status_z
      CHARACTER (*) :: file_dens, fmt_dens, status_dens
      INTEGER  Mmax
      PARAMETER (Mmax = 50)
      PARAMETER (file_z = 'data_z')
      PARAMETER (fmt_z = '(2F17.9)')
      PARAMETER (status_z = 'OLD')
      PARAMETER (file_dens = 'density.m')
      PARAMETER (fmt_dens = '(2X, E22.14)')
      PARAMETER (status_dens = 'REPLACE')
      END MODULE init_online

      MODULE init_out
      CHARACTER (*) :: file_out !, fmt_out, status_out
      PARAMETER (file_out = 'result.temp')
      END MODULE init_out

      MODULE coeffunc
      USE init_data1
      CONTAINS
!!
!! The coefficient functions for convection (or drift) term:
!!
      FUNCTION func_b(x,y,z)
      REAL*8   func_b(dimen)
      REAL*8   x, y, z
       func_b(1) = - DSIN(PI * x)
       func_b(2) = - DSIN(PI * y)
       func_b(3) = - DSIN(PI * z)
      END FUNCTION func_b
!!
!! The coefficient function for u (zero-th order term):
!!
      FUNCTION func_c(x,y,z)
      REAL*8   func_c, x, y, z, temp
       temp = DCOS(PI * x)
       temp = temp + DCOS(PI * y)
       temp = temp + DCOS(PI * z)
       temp = temp - PI
       func_c = PI * temp
      END FUNCTION func_c
!!
!! The force function f:
!!
!      FUNCTION func_f(t,x,y,z)
!      REAL*8   func_f, t, x, y, z, temp
!      END FUNCTION func_f

      FUNCTION func_u(t, x,y,z)
```

```
      REAL*8   func_u, t, x, y, z, temp
!      REAL*8,  PARAMETER :: PI = 3.14159265358979323
      INTRINSIC  DSIN, DEXP
       temp = DSIN(PI * x)
       temp = temp * DSIN(PI * y)
       temp = temp * DSIN(PI * z)
       func_u = temp * DEXP(-3D0/2*PI*PI*t)
!       func_u = temp * DEXP(-4*PI*PI*t)
      END FUNCTION func_u
      END MODULE coeffunc

      FUNCTION func_h(x)
      REAL*8  func_h(dim_ob)
      REAL*8  x(dimen)
      REAL*8  temp, temp1, temp2
      INTRINSIC  ASIN, ACOS, SQRT
      temp = x(1)**2 + x(2)**2
      temp1 = x(1) / SQRT(temp)
      temp2 = x(3) / SQRT(temp+x(3)**2)
      func_h(1) = ACOS(temp1)
      func_h(2) = ASIN(temp2)
      END FUNCTION func_h

      PROGRAM filter3adi
!!
!! Main program begins:
!!
      USE init_cpu
      USE init_data2
      USE init_dens
      USE init_noise
      USE init_online
      REAL*8  Z(dim_ob, Mmax)
      REAL*8  hh(dim_ob, 0:nx,0:ny,0:nz)
      REAL*8  p(0:1, 0:nx,0:ny,0:nz)
      REAL*8  u(0:nx,0:ny,0:nz, -Lx:Lx,-Ly:Ly,-Lz:Lz)
      REAL*8  pmax, pmin
      REAL*8  tempi, temp, tau
      REAL*8  tempijk(dim_ob), ob_var_inv(dim_ob,dim_ob)
      INTRINSIC  ABS, EXP, MAX, MIN !! DABS, DEXP, DMAX1
      INTRINSIC  MATMUL, DOT_PRODUCT

!      PRINT *, 'Reading data and initialization'
!      PRINT *, 'and off-line calculations, please wait ...'

      OPEN (UNIT=8, FILE=file_z, STATUS=status_z)
      READ (UNIT=8, FMT=fmt_z) ( (Z(i,m),i=1,dim_ob), m=1,Mmax)
      CLOSE (UNIT=8)

!      p(0, :,:,:) = func_p0((/xx,yy,zz/))
      DO i = 0, nx
         DO j = 0, ny
            DO k = 0, nz
               hh(:, i,j,k) = func_h( (/xx(i), yy(j), zz(k)/) )
               p(0, i,j,k) = func_p0( (/xx(i), yy(j), zz(k)/) )
            END DO
         END DO
      END DO

!      PRINT *, 'On-line calculations now ...'
      PRINT *, 'Mmax =', Mmax
      tau = 40                !! exp(-40) = 4.25e-18
!         tau = 14.14          !! = sqrt(2*100)
!         tau = 21.21          !! = sqrt(2*225)
```

```
      ob_var_inv = 0
      DO ii = 1, dim_ob
         temp = 1D0 / sigma(ii)
         ob_var_inv(ii,ii) = temp * temp
      END DO
      time_loop: DO  m = 0, Mmax-1
         m1 = m + 1
c      CALL off_line(xx,yy,zz, indx,indy,indz, u)
         pmax = 0.0
         pmin = 0.0
         p(1, :,:,:) = 0.0
!         p(m1, :,:,:) = 0.0
         DO ix = 0,nx
            DO jy = 0,ny
               DO kz = 0,nz
         tempijk = Z(:,m1) - hh(:,jx,jy,jz)
         tempi = DOT_PRODUCT(tempijk, MATMUL(ob_var_inv,tempijk)/2)
         IF(tempi .LE. tau) THEN
         tempi = EXP(-tempi)
         temp = u(ix,jy,kz)
!        p(m1, jx,jy,jz) = tempi * temp
         p(1, jx,jy,jz) = tempi * temp
         pmax = MAX(pmax, p(1, jx,jy,jz))
         pmin = MIN(pmin, p(1, jx,jy,jz))
         END IF
               END DO
            END DO
         END DO
         IF(pmin .LT. 0.0) THEN
            PRINT *, ' p_min < 0 ... '
            p(1,:,:,:) = p(1,:,:,:) - pmin
            pmax = pmax - pmin
         END IF
         IF(pmax .GT. 0.0)  THEN
            WHERE(p(1,:,:,:) > 0.0)          &
     &         p(1,:,:,:) = p(1,:,:,:) / pmax
!           ELSEWHERE
!           END WHERE
         END IF
         IF(m1 .LT. Mmax)  p(0,:,:,:) = p(1,:,:,:)
      END DO time_loop

      PRINT *, 'Saving results of the last step ...'
      OPEN (UNIT=9, FILE=file_dens, STATUS=status_dens)
      WRITE (UNIT=9, FMT='(5A)') 'p = ['
      DO i = 0, nx
         DO j = 0, ny
            DO k = 0, nz
               temp = p(1, i,j,k)
               IF(temp .GE. 0.99D-99) THEN
                  WRITE (UNIT=9, FMT=fmt_dens) temp
               ELSE
                  WRITE (UNIT=9, FMT='(4X,A3)') '0.0'
               END IF
            END DO
         END DO
      END DO
!     &     (((p(Mmax, i,j,k), i=0,nx), j=0,ny), k=0,nz)
      WRITE (UNIT=9, FMT='(2A)') '];'
      CLOSE (UNIT=9)

      CONTAINS
!!
!! The initial density function p0 is defined here:
```

```
 !!
      FUNCTION func_p0(x)
      REAL*8  func_p0, temp
      REAL*8, DIMENSION (dimen) :: x, temp0, temp1
      INTRINSIC  MATMUL, DOT_PRODUCT, EXP
      temp0 = x - xmean
      temp1 = MATMUL(var_inv, temp0)
!      temp = MATMUL(TRANSPOSE(temp0),temp1)
      temp = DOT_PRODUCT(temp0, temp1)
      func_p0 = EXP(-temp)
      END FUNCTION func_p0

      END PROGRAM filter3adi

      SUBROUTINE adi3d
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!       New ADI for convection-diffusion equations
!!       Chuanxia Rao
!!       May 1997
!!       Feb 1996
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!       SUBROUTINE solution(t_fin, u)
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!       New ADI solver for 3D convection-diffusion equations
!!       May 9, 1997
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
      USE init_data2
      USE coeffunc
      INTEGER  ktime, Kmax
      REAL*8  t_fin, time
      REAL*8  u(0:nx, 0:ny, 0:nz)
      REAL*8  ux(nx1), uy(ny1), uz(nz1)
      REAL*8  coef0a(dimen, nx1, ny1,nz1)
      REAL*8  coef0b(dimen, nx1, ny1,nz1)
      REAL*8  coef0c(dimen, nx1, ny1,nz1)
      REAL*8  coef1a(dimen, nx1, ny1,nz1)
      REAL*8  coef1b(dimen, nx1, ny1,nz1)
      REAL*8  coef1c(dimen, nx1, ny1,nz1)
      REAL*8  aax(nx2), bbx(nx1), ccx(nx2), ddx(nx1)
      REAL*8  aay(ny2), bby(ny1), ccy(ny2), ddy(ny1)
      REAL*8  aaz(nz2), bbz(nz1), ccz(nz2), ddz(nz1)
      REAL  cpu, tarray(2), DTIME
      EXTERNAL DTIME
      EXTERNAL coef
      INTRINSIC  IDINT
      INTENT(INOUT) :: t_fin, u
!      INTENT(OUT) :: u

      cpu = DTIME(tarray)
      CALL coef( coef0a,coef0b,coef0c, coef1a,coef1b,coef1c )

!      DO ix = 0, nx
!         xi = xx(ix)
!      DO jy = 0, ny
!         yj = yy(jy)
!      DO kz = 0, nz
!         zk = zz(kz)
!         u(ix,jy,kz) = func_u(t0, xi,yj,zk)
!      END DO
!      END DO
!      END DO
      cpu = DTIME(tarray)
      PRINT *, ' cpu for the coefficients and initialization = ', cpu
```

```
      Kmax = IDINT( (t_fin-t0)/dt )
      t_fin = t0 + dt*Kmax
      IF(Kmax .NE. mt) PRINT *, ' Check: t_final might be changed.'

      DO ktime = 1, Kmax
         time = t0 + dt*ktime
!!
!! 1. The explicit part, in alternating directions:
!!
         DO ix = 1,nx1
            xi = xx(ix)
         DO jy = 1,ny1
            yj = yy(jy)
         DO kz = 1,nz1
            uz(kz) = coef0a(3,ix,jy,kz) * u(ix,jy,kz-1)  &
     &              + coef0b(3,ix,jy,kz) * u(ix,jy,kz)    &
     &              + coef0c(3,ix,jy,kz) * u(ix,jy,kz+1)
         END DO
            u(ix, jy, 1:nz1) = uz
!            u(ix, jy, 0)  = func_u(time, xi, yj, zz(0))
!            u(ix, jy, nz) = func_u(time, xi, yj, zz(nz))
         END DO
         END DO

         DO kz = 1,nz1
            zk = zz(kz)
         DO ix = 1,nx1
            xi = xx(ix)
         DO jy = 1,ny1
            uy(jy) = coef0a(2,ix,jy,kz) * u(ix,jy-1,kz)  &
     &              + coef0b(2,ix,jy,kz) * u(ix,jy,kz)    &
     &              + coef0c(2,ix,jy,kz) * u(ix,jy+1,kz)
         END DO
            u(ix, 1:ny1, kz) = uy
!            u(ix, 0, kz) = func_u(time, xi, yy(0), zk)
!            u(ix, ny,kz) = func_u(time, xi, yy(ny),zk)
         END DO
         END DO

         DO jy = 1,ny1
            yj = yy(jy)
         DO kz = 1,nz1
            zk = zz(kz)
         DO ix = 1,nx1
            ux(ix) = coef0a(1,ix,jy,kz) * u(ix-1,jy,kz)  &
     &              + coef0b(1,ix,jy,kz) * u(ix,jy,kz)    &
     &              + coef0c(1,ix,jy,kz) * u(ix+1,jy,kz)
         END DO
            u(1:nx1, jy, kz) = ux
!            u(0, jy, kz) = func_u(time, xx(0), yj, zk)
!            u(nx,jy, kz) = func_u(time, xx(nx),yj, zk)
         END DO
         END DO
!          u = u + f(ktime, :,:,:)
!                + f(ktime, ix,jy,kz)

!!
!! 2. The implicit part, in alternating directions:
!!

         DO jy = 1,ny1
         DO kz = 1,nz1
            aax = coef1a(1, 2:nx1,jy,kz)
            bbx = coef1b(1, 1:nx1,jy,kz)
```

```
            ccx = coef1c(1, 1:nx2,jy,kz)
            ddx = u(1:nx1, jy, kz)
!            ddx(1) = ddx(1) - coef1a(1, 1,jy,kz)*u(0,jy,kz)
!            ddx(nx1) = ddx(nx1) - coef1c(1, nx1,jy,kz)*u(nx,jy,kz)
         CALL tridiag(nx1, aax, bbx, ccx, ddx, ux)
            u(1:nx1, jy, kz) = ux
         END DO
         END DO

!      IF(ktime .EQ. 1) PRINT *, ' one implicit step survived ...'

         DO kz = 1,nz1
         DO ix = 1,nx1
            aay = coef1a(2, ix,2:ny1,kz)
            bby = coef1b(2, ix,1:ny1,kz)
            ccy = coef1c(2, ix,1:ny2,kz)
            ddy = u(ix, 1:ny1, kz)
!            ddy(1) = ddy(1) - coef1a(2, ix,1,kz)*u(ix,0,kz)
!            ddy(ny1) = ddy(ny1) - coef1c(2, ix,ny1,kz)*u(ix,ny,kz)
         CALL tridiag(ny1, aay, bby, ccy, ddy, uy)
            u(ix, 1:ny1, kz) = uy
         END DO
         END DO

         DO ix = 1,nx1
         DO jy = 1,ny1
            aaz = coef1a(3, ix,jy,2:nz1)
            bbz = coef1b(3, ix,jy,1:nz1)
            ccz = coef1c(3, ix,jy,1:nz2)
            ddz = u(ix, jy, 1:ny1)
!            ddz(1) = ddz(1) - coef1a(3, ix,jy,1)*u(ix,jy,0)
!            ddz(nz1) = ddz(nz1) - coef1c(3, ix,jy,nz1)*u(ix,jy,nz)
         CALL tridiag(nz1, aaz, bbz, ccz, ddz, uz)
            u(ix, jy, 1:nz1) = uz
         END DO
         END DO

!         WHERE (u < 0.0D0)  u = 0.0D0
!         WHERE (u(:,:,:) < 0.0D0)  u(:, :, :) = 0.0D0

      END DO

      RETURN
      END SUBROUTINE adi3d
!      END SUBROUTINE solution

      SUBROUTINE coef( coef0a,coef0b,coef0c, coef1a,coef1b,coef1c )
!      SUBROUTINE coef( coef0a,coef0b,coef0c, coef1a,coef1b,coef1c, f )
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!      The coefficients for solving the 1D sub-problems
!!      May 9, 1997
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
      USE init_data2
      USE coeffunc
      REAL*8  coef0a(dimen, nx1, ny1,nz1)
      REAL*8  coef0b(dimen, nx1, ny1,nz1)
      REAL*8  coef0c(dimen, nx1, ny1,nz1)
      REAL*8  coef1a(dimen, nx1, ny1,nz1)
      REAL*8  coef1b(dimen, nx1, ny1,nz1)
      REAL*8  coef1c(dimen, nx1, ny1,nz1)
!      REAL*8  f(0:nx, 0:ny, 0:nz)
      REAL*8  dxyz(dimen), dneg(dimen), dpos(dimen)
      REAL*8  dxyz_neg, temp, temp_c, vect_b(dimen)
      LOGICAL judge(dimen)
```

```
!      COMMON  /coef_right/  convection, diffusion, dxyz
      INTRINSIC  DABS, DSIN, DCOS
      INTENT(OUT) ::  coef0a,coef0b,coef0c, coef1a,coef1b,coef1c

      DO i = 1, dimen
         dxyz(i) = diffus(i)*dtx2(i)/dx(i)
         temp = 2 * dxyz(i)
         dneg(i) = 1 - temp
         dpos(i) = 1 + temp
      END DO

      DO ix = 1, nx1
         xi = xx(ix)
      DO jy = 1, ny1
         yj = yy(jy)
      DO kz = 1, nz1
         zk = zz(kz)

         vect_b = func_b( xi, yj, zk )
         temp_c = func_c( xi, yj, zk ) * dtdim2
         judge = (vect_b .LE. 0D0)

         DO i = 1, dimen
            temp = vect_b(i) * dtx2(i)
            temp_b = DABS(temp)
            dxyz_neg = - dxyz(i)
         IF (judge(i)) THEN
            coef0a(i, ix,jy,kz) = dxyz(i)
            coef0c(i, ix,jy,kz) = dxyz(i) + temp
            coef1a(i, ix,jy,kz) = dxyz_neg + temp
            coef1c(i, ix,jy,kz) = dxyz_neg
         ELSE
            coef0a(i, ix,jy,kz) = dxyz(i) - temp
            coef0c(i, ix,jy,kz) = dxyz(i)
            coef1a(i, ix,jy,kz) = dxyz_neg
            coef1c(i, ix,jy,kz) = dxyz_neg - temp
         END IF
            coef0b(i, ix,jy,kz) = dneg(i) +  temp_b + temp_c
            coef1b(i, ix,jy,kz) = dpos(i) +  temp_b - temp_c
         END DO

      END DO
      END DO
      END DO

      RETURN
      END SUBROUTINE coef

!      SUBROUTINE f_add(vv, jjudge, dd_judge,dd, ff)
!      REAL*8  vv, dd_judge, dd, ff
!      LOGICAL jjudge
!      INTENT(IN) :: vv, jjudge, dd_judge, dd
!      INTENT(INOUT) :: ff
!      IF(vv .NE. 0D0) THEN
!         IF(jjudge) THEN
!            ff = ff + vv * dd_judge
!         ELSE
!            ff = ff + vv * dd
!         END IF
!      END IF
!      RETURN
!      END SUBROUTINE f_add

      SUBROUTINE tridiag(N,A,B,C,D,X)
```

```
      INTEGER  N
      REAL*8  A(N-1),B(N),C(N-1),D(N),X(N)
      INTENT(IN) :: N, A,C
      INTENT(INOUT) :: B,D, X
      DO 2 I = 2,N
        XMULT = A(I-1)/B(I-1)
        B(I) = B(I) - XMULT*C(I-1)
        D(I) = D(I) - XMULT*D(I-1)
 2    CONTINUE
      X(N) = D(N)/B(N)
      DO 3 I = N-1,1,-1
        X(I) = (D(I) - C(I)*X(I+1))/B(I)
 3    CONTINUE
      RETURN
      END SUBROUTINE tridiag
```